UNIVERSITÀ DI PISA **FACOLTÀ DI INGEGNERIA**

CORSO di LAUREA IN INGEGNERIA INFORMATICA

Progettazione ed implementazione di una libreria C++ per il supporto alla programmazione di sistemi in tempo reale

Anno Accademico 2010/2011

Relatori:

prof. Marco Avvenuti
Dip. Ingegneria dell'Informazione

prof. Giuseppe Lipari
Scuola Superiore S. Anna

prof. Giuseppe Anastasi
Dip. Ingegneria dell'Informazione

Candidato:

Silvano Marchetti

Sommario

Introduzione
e obiettivi della tesi

I sistemi multiprocessore sono l'attuale architettura predominante su cui si basano i sistemi informatici. I sistemi multi-core sono attualmente utilizzati nell'ambito dei sistemi embedded, della telefonia mobile (smart phone), server, etc.

Questa tendenza deriva dalla maggiore difficoltà nell'aumentare la frequenza di clock di un singolo processore rispetto a quella di far collaborare insieme più unità. Raddoppiare la velocità o raddoppiare il numero di sistemi porta a circa lo stesso risultato operativo, ma il secondo comporta molti meno problemi e costi in termini di consumo energetico, surriscaldamento, etc.

Nell'ambito dei sistemi multiprocessore abbiamo deciso di analizzare lo sviluppo che questi hanno avuto nel campo dei sistemi real-time: questi trovano un'ampia applicazione nel settore industriale e del controllo, in modo particolare quando dobbiamo ottenere una risposta dal sistema entro un tempo ben definito. Un sistema real-time non dovrà comunque essere necessariamente veloce, l'obbligo stringente è che il sistema risponda entro un tempo massimo prefissato. In un sistema real-time devo cioè garantire che un'elaborazione termini entro un determinato vincolo temporale detto *deadline*.

Parallelamente allo sviluppo dei sistemi real-time abbiamo notato l'affermarsi delle librerie C++ Boost: una collezione di librerie open source per la programmazione multithread che estendono le funzionalità del C++ e che sono state accettate per essere incorporate nel nuovo standard del C++ che uscirà prossimamente.

Nello specifico abbiamo quindi analizzato la *boost::thread* e abbiamo notato come questa fornisca ben poche funzionalità nello sviluppo dei thread in ambito real-time; abbiamo quindi deciso di sviluppare una nostra libreria dove estenderemo le

funzionalità della boost::thread alle applicazioni real-time periodiche e non, e dove inseriremo, basandoci sempre sulla Boost, delle utility che sono risultate molto comode nella fase di test dei thread stessi.

Come requisiti ci siamo imposti di creare una libreria Object Oriented e che questa mantenesse la portabilità sulle varie piattaforme (Windows, Linux, ecc.) o quantomeno la facile adattabilità.

Per la fase di test, del resto molto importante in special modo per i sistemi real-time, ci appoggeremo alla libreria Google Test: una libreria anch'essa utilizzabile su una varietà di piattaforme (Linux, Mac, Windows,….), e che ci permetterà di controllare la correttezza del requisiti di ogni funzione da noi realizzata, rendendo i test visibili esclusivamente nella sola fase di test. Grazie al suo metodo implementativo infatti, il codice della libreria e quello della fase di test rimarranno totalmente separati.

Successivamente abbiamo sviluppato un meccanismo per la gestione dei cambi di modo di lavoro dei task, anche questa, una delle realtà implementative attualmente in fase di studio, e che riguarda la gestione e il controllo delle varie modalità operative nei sistemi real-time.

La nostra trattazione sarà quindi così articolata:

CAP. 1: I Sistemi Real Time, una trattazione generica per capire cosa sono i sistemi realtà, le terminologie, i metodi di schedulazione e la gestione dei cambi di modo dei task.

CAP. 2: La Libreria pthread, chiariremo perché è più conveniente utilizzare un thread rispetto ad un Processo, faremo inoltre una panoramica delle funzionalità della libreria posix pthread su cui ci baseremo per la modifica della priorità e della politica di schedulazione dei thread.

CAP. 3: Le Librerie Boost, tratteremo in modo particolare la boost::thread ma indicheremo anche alcune funzionalità extra che utilizzeremo per il nostro sviluppo.

CAP. 4: Google Test, un piccolo richiamo delle funzioni base della libreria che utilizzeremo nella fase di test di tutte le classi e le funzioni da noi sviluppate.

CAP. 5: Progettazione Libreria, contiene la descrizione e la modalità di utilizzo della libreria sviluppata per mezzo di modelli, diagrammi UML e frammenti di codice. Per una descrizione più dettagliata, il codice, approfondimenti e sviluppi futuri del progetto rimandiamo la trattazione al repository da noi utilizzato: http://sourceforge.net/projects/threadutility/

CAP. 6: Esperimenti, presenteremo un software che manda in esecuzione 5 thread in modalità real-time su sistema linux con patch real-time, lo faremo sia

utilizzano la nostra libreria, sia utilizzando esclusivamente le capacità offerteci dalla boost::thread e dalla pthread; valuteremo quindi le prestazioni delle due implementazioni andando ad analizzare la traccia di utilizzo del kernel per mezzo della KernelShark. Un ulteriore esperimento, utilizzando la KernelShark sulla traccia del Kernel, è un esempio applicativo dei cambi di modo.

CAP. 7: Conclusioni, a seguito della fase di progetto, lo sviluppo e gli esperimenti fatti ci permetteremo di esprimere un modesto parere sul lavoro e i risultati ottenuti.

Cap. 1 - I Sistemi Real-Time

1.1. Introduzione

Innanzitutto chiariamo *cosa si intende per Sistema Real-Time (Tempo Reale).*

Un Sistema Real-Time è un sistema che controlla un sistema fisico che a sua volta interagisce con l'ambiente a questo esterno.

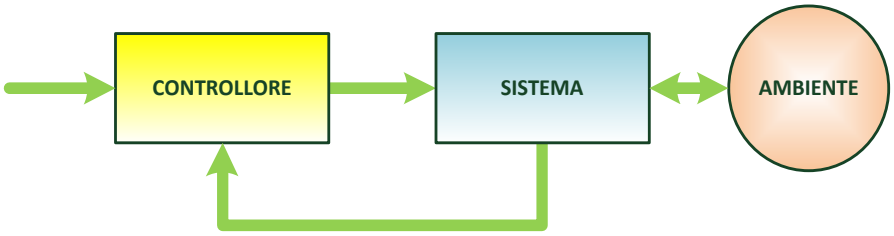
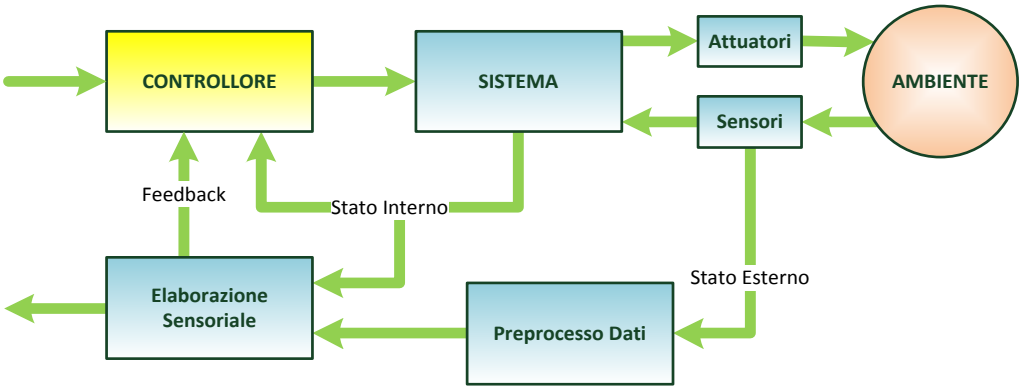


Figura 1. Modello di Sistema Real-Time che interagisce con l'Ambiente.

Le variabili che caratterizzano il sistema fisico saranno in continua evoluzione, il sistema real-time le controllerà reazionandole in modo da far evolvere il sistema nel modo desideriamo.

Dobbiamo spendere due parole sul fattore tempo, un concetto di fondamentale importanza per un Sistema Real-Time. ***In un Sistema Real-Time infatti la correttezza del risultato non dipende solo dai dati, ma anche dal tempo con cui questo viene fornito.*** Un risultato corretto ma che arriva in ritardo infatti può risultare dannoso per il sistema.



La velocità è un concetto relativo, l'efficienza di un Sistema Real-Time non dipende dalla velocità, in quanto la sua velocità dipende dall'ambiente con cui il sistema interagisce.

Un sistema veloce privilegia le prestazioni medie, un Sistema Real-Time privilegia le prestazioni individuali, in quando dobbiamo garantire ad ogni processo che terminerà entro un certo istante temporale.

Alcuni esempi di Sistemi che necessitano di controlli Real-Time sono:

- le centrali chimiche-nucleari
- i sistemi militari
- il controllo degli aerei
- il monitoraggio del traffico
- i problemi di robotica

Se dobbiamo gestire una sola attività e abbiamo dei problemi sul tempo di risposta potremmo pensare di utilizzare un calcolatore più veloce; se invece ci sono più attività da gestire in modalità Real-Time dovremo concepire il sistema in modo che gestisca la sincronizzazione tra le varie attività.

Nel passato la progettazione di questi sistemi veniva fatta in modo empirico, ottimizzando il più possibile il codice e scrivendone alcune porzioni completamente in Assembler.

Nei sistemi real-time è molto importante il *concetto di correttezza nella progettazione del software*: gestendo in alcuni casi sistemi di elevata pericolosità, un eventuale errore che si presentasse in fase di esecuzione potrebbe causare dei seri problemi anche a livello di sicurezza. Parallelamente alla progettazione del software risulta molto importanti la **fase di analisi e test:** durante la fase di progetto dovremo prevedere tutte le possibili situazioni critiche che si potrebbero presentare e dovremo progettare i sistemi in modo tale che supportino i carichi di picco. Dovremo sempre prevedere che il sistema possa trovarsi nella peggiore delle situazioni possibili.

1.2. Definizioni e Terminologie

Prima di scendere nel dettaglio dei Sistemi Real-Time vediamo qualche richiamo e qualche definizione.

1.2.1. Processo o Task

Si dice **Processo** o **Task** un'istanza di un programma che è in esecuzione in modo sequenziale. Questa attività sarà controllata da un programma, in genere gestito dal rispettivo sistema operativo, che si svolge su un processore. Un'attività svolta da un processo in genere viene eseguita in modo continuativo fino al suo completamento a patto che non ci siano interruzioni da parte di altri attività in corso sul medesimo processore.

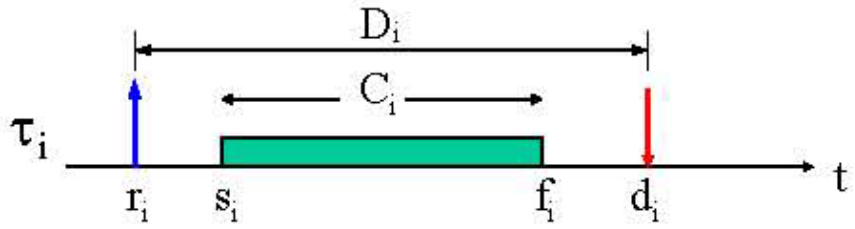


Figura 2. Time Line di un Processo

Un Processo può quindi essere caratterizzato dai seguenti parametri:

- τ_i – processo i-esimo dell'applicazione
- r_i – *"Request Time"*, tempo di arrivo della richiesta
- d_i – *"DeadLine"*, scadenza, tempo entro il quale il processo deve terminare
- s_i – *"Start Time"* – primo istante in cui il processo entra in esecuzione
- f_i – *"Finishing Time"* – effettiva terminazione dell'attività di elaborazione di quel processo
- c_i – *"Computation Time"* – tempo totale di elaborazione del processo
- D_i – *"DeadLine Relativa"* – è la deadline relativa al tempo di partenza del processo:
  $D_i = d_i - r_i$
- R_i – *"Response Time"* – tempo di risposta relativo alla richiesta

Un Processo è un'entità utilizzata dal sistema operativo per rappresentare una specifica esecuzione di un programma[1]. Si tratta quindi di un'entità dinamica che evolve in base ai dati elaborati e alle operazioni eseguite su di essi.

Molto spesso si utilizza il termine **Thread** in sostituzione di processo, i due concetti sono simili ma distinti. Per processo si intende un oggetto del sistema operativo a cui sono assegnate tutte le risorse di sistema per l'esecuzione di un programma, tranne la CPU; un thread invece è un oggetto del sistema operativo a cui viene assegnata la CPU per l'esecuzione. *L'esecuzione di un processo può infatti essere suddivisa in più thread* che eseguono a divisione di tempo o in parallelo ad altri thread sul medesimo processore, il Thread è cioè una parte del processo che viene eseguita in modo concorrente. Un'altra differenza sostanziale tra Processi e Thread consiste nel modo di condivisione delle risorse, mentre i processi sono solitamente indipendenti e quindi per comunicare ed interagire devono utilizzare degli appositi meccanismi di comunicazione messi a disposizione dal sistema, i Thread invece condividono le medesime informazioni di stato, la memoria e le risorse del sistema. Ne deriva quindi che l'istanza di un Processo è più onerosa rispetto a quella di un Thread.

1.2.2. Stati di un Processo

In un Sistema Operativo Multitasking, vengono eseguiti contemporaneamente diversi processi, questi potranno utilizzare il processore per un periodo di tempo limitato (a divisione di tempo); per questo motivo i processi vengono interrotti, messi in pausa e richiamati in base a quanto specificato dagli algoritmi di schedulazione, questo comportamento fa sembrare all'utente che i processi vengano eseguiti in modo parallelo.

[1] Un *programma* è costituito dal codice eseguibile che si ottiene dalla compilazione del codice sorgente, ed è in genere salvato sotto la forma di uno o più file. Questa è un'entità statica che quindi non cambia durante l'esecuzione.
[2] *Quanto Temporale*
[3] Intervallo di tempo che individua la risoluzione del sistema
[4] Si dice *"Overhead"* il tempo impiegato dal processore per gestire i meccanismi interni al nucleo.
[5] Per *"Jitter"* si intende il ballio all'interno.

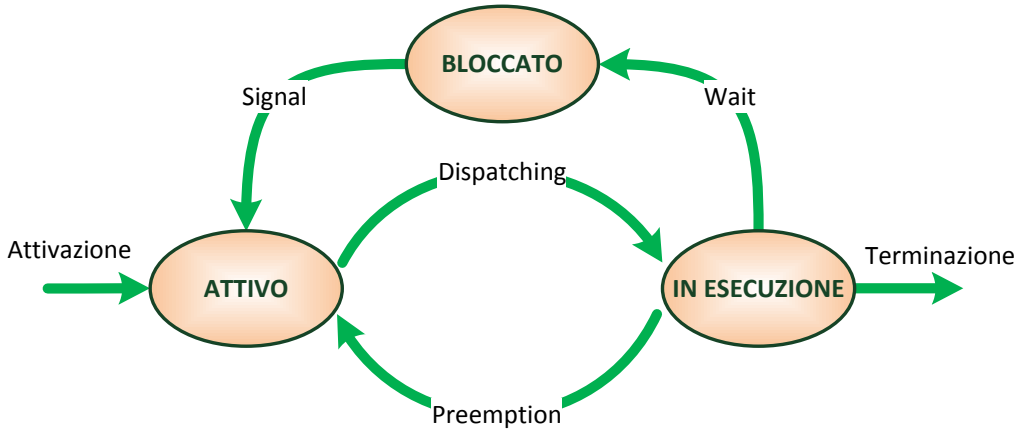


Figura 3. Stati di un Processo con schedulazione Preemptive.

Un processo quindi durante la propria esecuzione può trovarsi nello stato **Attivo** se può essere eseguito dalla CPU, **Bloccato** se è in attesa di un evento che lo sblocchi. Se un processo è Attivo, questo potrà essere **In Esecuzione** se sta usando la CPU, o **Pronto** se è in attesa dell'uso della CPU. Un processo quindi quando viene attivato (*Activate*) entra nella coda dei processi pronti (*Ready Queue*). Al momento della commutazione di contesto viene eseguita l'operazione di *dispatching* con cui il Sistema assegna il processore al processo pronto di maggiore priorità. A questo punto il processo entra in esecuzione. Il processo potrà tornare nella coda dei processi pronti a causa del *preemption* (se attivo), cioè se in coda dei pronti è presente un processo con priorità più alta di quello che è attualmente in esecuzione. Se invece l'algoritmo di schedulazione *non è preemptive* vuol dire che il Processo che è in esecuzione non può essere sospeso finché non è stato completato.

1.2.3. Schedulazione

Un algoritmo di schedulazione non è altro che il criterio con cui un task viene assegnato al processore.

Come già detto, un algoritmo di schedulazione si dice:

- **preemptive**: se i task in esecuzione possono essere temporaneamente sospesi e reinseriti nella coda dei processi pronti in modo da mandare in esecuzione i task più importanti
- **non preemptive**: se i task in esecuzione non possono essere sospesi fino a quando non sono ultimati

Nei seguenti grafici possiamo vedere un esempio di schedulazione con e senza preemption

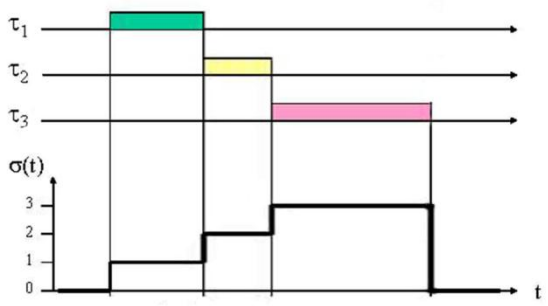


Figura 4. Schedulazione non Preemptive.

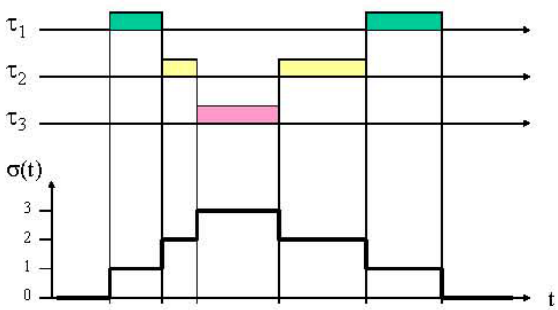


Figura 5. Schedulazione Preemptive.

Una corretta schedulazione prevede che vengano rispettati i vincoli imposti dalla deadline, in caso di superamento della deadline da parte di un processo (*deadline miss*) possiamo incorrere in problematiche differenti a seconda che si tratti di Task Hard (possono verificarsi anche degli effetti catastrofici) o di Task Soft (in questo caso il superamento della deadline causa solo il degrado delle prestazioni).

1.2.4. Modalità di Attivazione

Si dice ***Task Periodico***, un processo che viene attivato automaticamente dal sistema ad intervalli regolari.

Si dice ***Task Aperiodico***, un processo che si attiva all'arrivo di un evento oppure mediante l'invocazione esplicita della primitiva di attivazione.

1.2.5. Tipi di Vincoli

Si distinguono varie tipologie di vincoli a cui i task devono sottostare:

- **Vincoli Temporali:** che impongono le condizioni sull'attivazioni e la terminazioni dei Task, nel caso specifico se imponiamo che un processo termini entro una certa deadline, la condizione potrà essere di tipo:
  - **Hard:** se si vuole che tassativamente la computazione termini entro la deadline in quando una terminazione fuori deadline causerebbe un fallimento dell'intero sistema
  - **Soft:** se la condizione di terminazione è desiderato che avvenga entro la deadline ma in caso di fallimento possiamo continuare l'esecuzione
- **Vincoli di Precedenza:** che impongono un ordinamento nell'esecuzione dei Task che devono essere eseguiti rispettando delle relazioni di precedenza specificate da un Grafo Diretto Aciclico
- **Vincolo su Risorse:** che impongono una sincronizzazione nell'accesso alle risorse mutuamente esclusive: infatti affinché i dati rimangano consistenti, le risorse condivise devono essere accedute in mutua esclusione.

1.2.6. Schedulabilità

Si dice che una *schedulazione è fattibile* se tutti i task completano la loro esecuzione rispettando un insieme di vincoli specificati. Un insieme di task G si dice *schedulabile* se

per esso esiste una schedulazione fattibile.

Definizione:
Dato un insieme **Γ** di ***n*** task, un insieme **P** di ***m*** processori, e un insieme **R** di ***r*** risorse, dobbiamo trovare un assegnamento di **P** ed **R** a **Γ** che produca una schedulazione fattibile.

Garey e Johnson nel 1975 hanno dimostrato che il problema della schedulabilità fattibile ha una complessità non polinomiale, a patto che sui sistemi di gestione e sui task considerati vengano rispettate alcune condizioni semplificative:

- si considera un unico processore
- si ha un insieme di task omogenei tra loro
- i task sono preemptive
- le attivazioni sono simultanee
- non ci sono vincoli di precedenza
- non ci sono vincoli sulle risorse.

La schedulazione può essere di tipo **Statico** se le decisioni di scheduling dipendono da parametri fissi assegnati staticamente ai task prima dell'attivazione; oppure **Dinamico** se le decisioni di scheduling dipendono da parametri che variano nel tempo.

1.3. Algoritmi di Schedulazione per sistemi Non Real-Time

1.3.1. Round Robin(RR)

Consideriamo di suddividere l'asse temporale a fette, si assegna un quanto di tempo a ciascun processo. I processi vengono quindi eseguiti un pezzetto per volta e si susseguono l'un l'altro in modo ciclico. Ogni task τ_i quindi non può rimanere in esecuzione per più di Q[2] unità di tempo, esaurito il Quanto Temporale, il processo τ_i che era in esecuzione viene reinserito alla fine della coda dei processi pronti e viene mandato in esecuzione il primo elemento della coda. La coda dove vengono immagazzinati i processi attivi viene gestita in modalità FIFO.

Se n è il numero di task del sistema, un processo esegue un pezzetto ogni n quanti di tempo, quindi ogni $n \cdot Q$ secondi.

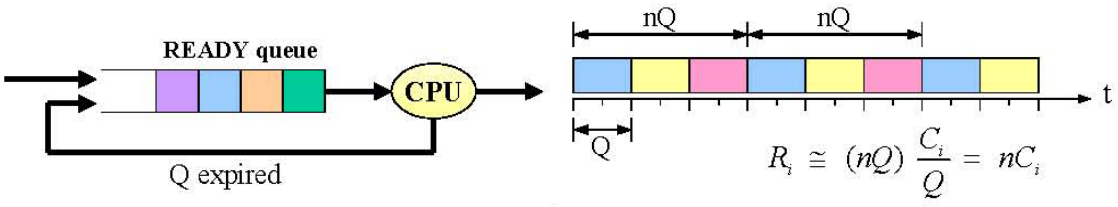


Figura 6. Esempio di Schedulazione Round Robin.

[2] *Quanto Temporale*

Se l'i-esimo processo viene suddiviso in $\frac{C_i}{Q}$ pezzetti, dove C_i è il computation time; allora il processo terminerà entro $f_i = \frac{C_i}{Q} \cdot n \cdot Q = n \cdot C_i$ secondi.

Ma quale dimensione deve avere il quando temporale ?

Se $Q > max(C_i)$ allora tutti i task terminano entro il primo quando e l'algoritmo Round Robin equivale al First Come First Served. Se invece il quanto Q è piccolo e paragonabile al tempo di commutazione di contesto (δ), allora lo schema temporale visto deve essere modificato.

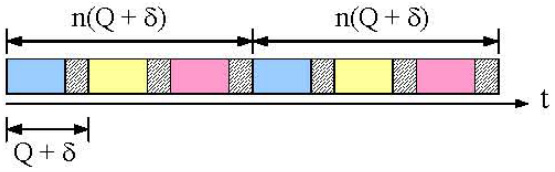


Figura 7. Schedulazione Round Robin con quanto Q piccolo e paragonabile al cambio di contesto δ.

In questo caso l'i-esimo processo termina entro $f_i = \frac{C_i}{Q} \cdot n \cdot (Q + \delta) = n \cdot C_i \cdot \left(\frac{Q+\delta}{Q}\right)$, ciò in quanto il tempo di commutazione non è trascurabile. Se $Q = \delta$ allora $f_i = 2 \cdot n \cdot C_i$.

Anche questo algoritmo non può però essere utilizzato nei sistemi real-time in quanto non tiene conto del fatto che un processo può avere "più fretta".

1.4. Algoritmi di Schedulazione per sistemi Real-Time

1.4.1. Algoritmo Prioritario

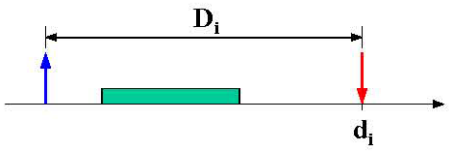
Con questo algoritmo assegniamo una priorità p_i a ogni processo e i vari processi vengono mandati in esecuzione in base alla loro priorità.

Con questo algoritmo c'è il rischio di incorrere nel problema dello ***starvation***: i task con priorità molto bassa prima di andare in esecuzione rischiano di rimanere in attesa per tempi molto lunghi. Questo algoritmo quindi penalizza i processi a bassa priorità.

- Se poniamo $p_i = \frac{1}{r_i}$ l'algoritmo prioritario si riduce all'algoritmo "First Come First Serve".
- Se poniamo $p_i = \frac{1}{c_i}$ l'algoritmo prioritario si riduce all'algoritmo "Shortest Job First".

Negli Algoritmi Real-Time i task possono essere schedulati in base a:

- deadline relativa D_i
- deadline assoluta d_i



1.4.2. Earliest DeadLine First – EDF

In questo caso i Task arrivano in modo dinamico, i processi si attivano cioè in istanti diversi $r_i \neq r_j$, $\forall\, i \neq j$ inoltre la schedulazione viene eseguita in base alla deadline assoluta e non relativa.

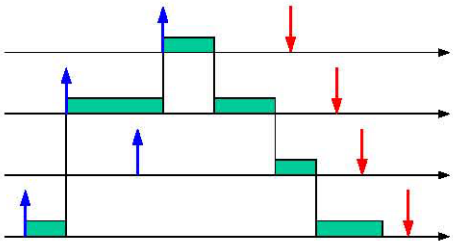


Figura 8. Schedulazione EDF.

Si dice che *l'algoritmo EDF è ottimo nel senso della schedulabilità*, se esiste cioè una schedulazione fattibile, l'algoritmo EDF la trova. Se l'algoritmo EDF non è in grado di generare una schedulazione fattibile, allora nessun altro algoritmo è in grado di generarla.

Poiché abbiamo detto che è previsto l'arrivo dinamico dei task *dobbiamo garantire dinamicamente la schedulabilità del sistema all'arrivo di ciascun nuovo task.* Supponendo cioè che il sistema $\mathcal{T}$ sia schedulabile, vogliamo sapere se il sistema $\mathcal{T}' = \mathcal{T} \cup \{t_{new}\}$ è schedulabile.

Introduciamo il concetto di *Tempo Residuo di Calcolo* (c_i)*;* poiché sappiamo che in questo algoritmo la priorità di un processo è inversamente proporzionale alla sua deadline assoluta $\left(p_i \propto {}^{1}/_{d_i}\right)$, il tempo di fine di un processo t_i sarà $f_i = t + \sum_{k:d_k<d_i}^{i} c_k$, cioè l'istante corrente *t* più i tempi residui di calcolo di tutti i processi con priorità superiore a p_i . In questo modo all'arrivo di un nuovo task possiamo determinare se il nuovo sistema è schedulabile o meno.

1.5. Algoritmi di Schedulazione per Sistemi con Task Periodici

Gli algoritmi che abbiamo analizzato fino ad ora non sono in grado di gestire i processi che vengono attivati periodicamente.

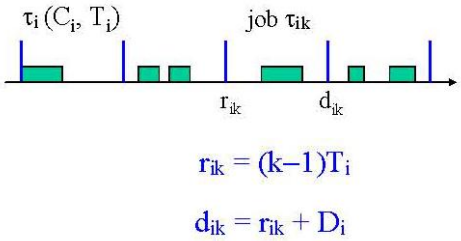


Figura 9. Modello rappresentativo di Task Periodico.

Un processo quando viene attivato viene posto nella coda dei processi pronti ed entra in esecuzione quando viene scelto dall'algoritmo di schedulazione. Se subisce

preemption il processo tornerà nella coda dei pronti ma tornerà in esecuzione appena lo schedulatore lo sceglierà nuovamente. Non appena viene interrotto il processo entra nello stato ***idle*** e vi rimarrà in attesa di essere risvegliato. La sospensione e la riattivazione dei processi vengono invocate dalle primitive END_ISTANCE e WAKE_UP rispettivamente.

Ad ogni TICK[3] il sistema si ferma per effettuare dei controlli: tra queste operazioni c'è anche quella di controllare se nella coda "idle" ci sono dei processi che devono essere riattivati.

1.5.1. TimeLine Scheduling

Questo è un algoritmo di schedulazione particolarmente affidabile anche se empirico.

Suddividiamo il tempo in intervalli di lunghezza costante Δ. Questo intervallo di tempo deve essere il più grande possibile continuando però a mantenere i rispettivi vincoli. In genere si sceglie il massimo comune divisore dei periodi che devono essere rispettati:

$$\Delta = MCD(p1, p2, \dots, pn) \qquad \textit{(MINOR CYCLE)}$$

La schedulazione che effettueremo si ripeterà ogni t_M pari al minimo comune multiplo dei periodi:

$$t_M = mcm(p1, p2, \dots, pn) \qquad \textit{(MAJOR CYCLE)}$$

A questo punto possiamo costruire empiricamente la schedulazione inserendo per primo nel minor cycle il processo con periodo più piccolo; successivamente cercheremo di inserire tutti gli altri processi procedendo da quelli più ricorrenti (con periodo minore) a quelli meno ricorrenti (con periodo maggiore).

Per implementare un sistema di questo tipo è necessario utilizzare un timer che invia un segnale di sincronizzazione ogni Δ secondi.

Vediamo un Esempio:

Consideriamo tre processi periodici con le seguenti caratteristiche

Attività	A	B	C
Periodo	25 ms	50 ms	100 ms

$$\Delta = MCD(25, 50, 100) = 25 \text{ ms} \qquad t_M = mcm(25, 50, 100) = 100\ ms$$

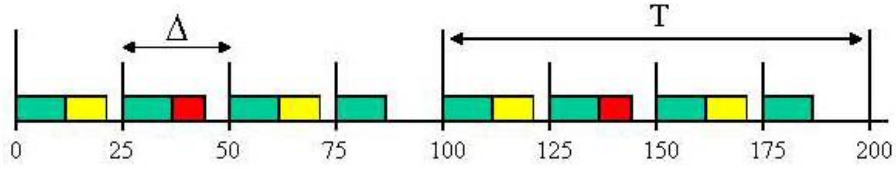


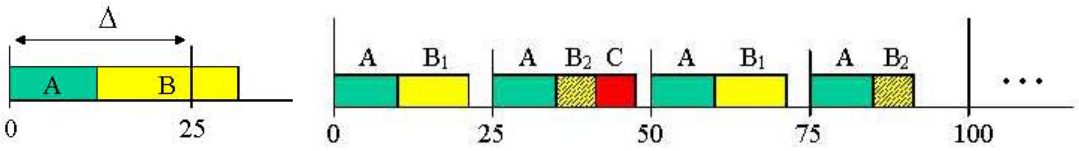
Figura 10. Esempio di Schedulazione di Tipo TimeLine.

[3] Intervallo di tempo che individua la risoluzione del sistema

Questa schedulazione risulta molto semplice e fornisce una visione completa della situazione. Inoltre avendo poche commutazioni di contesto, abbiamo un **basso over-head**[4] e come abbiamo visto, l'implementazione di questa schedulazione è molto semplice e non implica l'utilizzo di nessun particolare sistema operativo. Infine anche il **Jitter**[5] **è ridotto.**

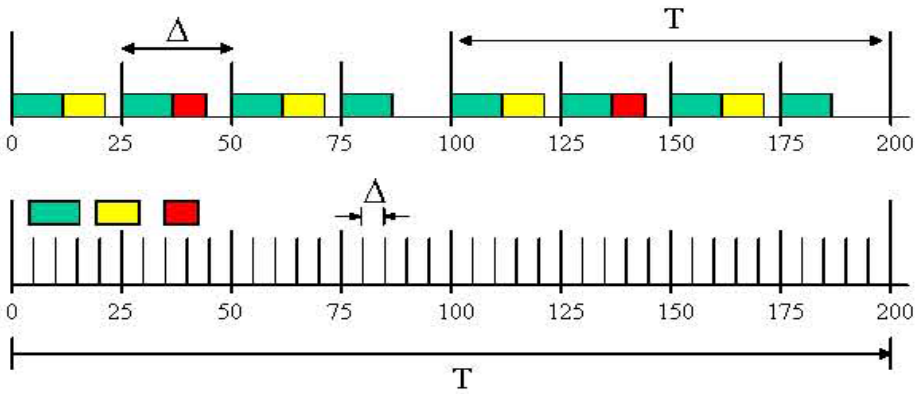
Questa tipologia di schedulazione ha insiti però due svantaggi:

1. **Il sistema risulta fragile nei sovraccarichi transitori**
   Considerando il precedente esempio, per garantire la schedulazione bisogna che:
   $$C_A + C_B \leq \Delta \qquad \text{e} \qquad C_A + C_C \leq \Delta$$
   Ci potrebbe però essere un caso di OverRun, può cioè accadere che una routine continui oltre il tempo previsto. Potrebbe quindi accadere che quando arriva il segnale da parte del timer sia ancora in esecuzione una procedura, in questi casi possiamo **lasciar finire** il processo in esecuzione creando un effetto domino sulle altre esecuzioni, oppure **abortire** la procedura rischiando di lasciare le strutture dati in uno stato inconsistente.
2. **Il sistema risulta difficilmente espandibile**
   Nel caso in cui si debba aggiungere un'attività D, oppure nel caso in cui l'attività B venga trasformata in B' con ($C_{B'} > C_B$), può verificarsi che un'attività debba essere spezzata in più parti. Ciò non risulta sempre facile da fare.



Inoltre potrebbe cambiare la frequenza con cui l'attività deve essere eseguita. Se ad esempio l'attività B dovesse passare ad essere eseguita ogni 40 ms, avremmo i seguenti cambiamenti:

$$\Delta = MCD(25, 40, 100) = 5 \text{ ms} \qquad t_M = mcm(25, 40, 100) = 200\, ms$$



[4] Si dice *"Overhead"* il tempo impiegato dal processore per gestire i meccanismi interni al nucleo.
[5] Per *"Jitter"* si intende il ballio all'interno.

In questo modo abbiamo dovuto riprogettare tutto il sistema. Questa serie di problemi viene risolto ricorrendo ad algoritmi che lavorano su base prioritaria.

1.6. Schedulazione Prioritaria per Task Periodici

Abbiamo visto con il precedente esempio che se i periodi non sono "armonici" otteniamo un minor cycle troppo piccolo e ciò determinerebbe uno spezzettamento delle attività in troppi pezzi. Questo problema si risolve utilizzando gli algoritmi di schedulazione a basse prioritaria.

La priorità dei vari processi può essere impostata secondo due regole:

1. **Regola Statica:** dove la priorità dei processi risulta essere inversamente proporzionale al periodo.
   $p_i \propto \frac{1}{T_i}$ è questo il caso dell'algoritmo *Rate Monotonic – RM*
2. **Regola Dinamica:** dove la priorità dei processi risulta essere inversamente proporzionale alla dead-line assoluta.
   $p_i \propto \frac{1}{d_i}$ è questo il caso dell'algoritmo *Earliest Dead First – EDF*

Si definisce **Iperperiodo H** il minimo comune multiplo dei periodi di un insieme di task.

$$H = mcm\{T_i\}$$

Se tutti i task arrivano nello stesso istante, dopo ogni iperperiodo la schedulazione si ripete esattamente.

1.6.1. Algoritmo Rate Monotonic

Si tratta di un algoritmo prioritario dove la priorità di ogni processo viene imposta come l'inverso del periodo del processo stesso.

Vediamo un Esempio:

Attività	A	B	C
Computation Time	10	10	20
Periodo	25 ms	40 ms	100 ms

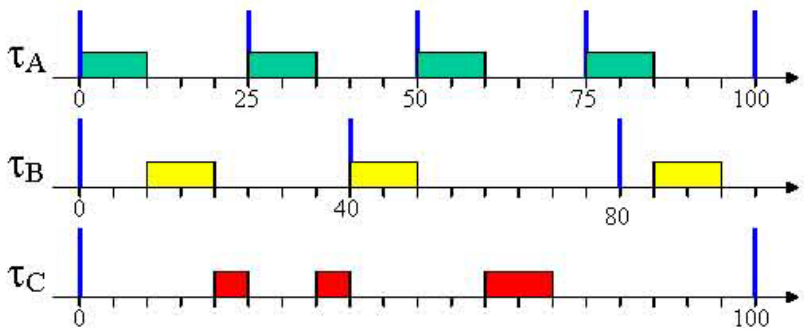


Figura 11. Esempio di Schedulazione Rate Monotonic.

In questo modo non abbiamo dovuto spezzettare i vari task, ma come possiamo garantire a priori che tutto il sistema risulti schedulabile? Si può fare con *l'analisi di Liu e Layland.*

Analisi Liu e Layland ('73)

Supponiamo che ogni processo τ_i sia in esecuzione per un tempo C_i ogni periodo T_i.

Possiamo definire un parametro che quantifica l'UTILIZZAZIONE DEL PROCESSO: $U_i \triangleq \frac{C_i}{T_i}$.

Quindi un insieme di *n* task determinerà un carico periodico detto **FATTORE DI UTILIZZAZIONE** TOTALE pari a:

$$U_p \triangleq \sum_i U_i = \sum_i \frac{C_i}{T_i}$$

Detto ciò possiamo affermare che **Condizione Necessaria, ma NON Sufficiente** affinché un insieme di Task risulti **Schedulabile** è che il fattore di utilizzazione totale risulti minore di 1:

$$U_p < 1$$

La condizione di non sufficienza la possiamo verificare con i seguenti esempi dove nonostante la condizione sia rispettata non sempre si ha schedulabilità:

1. In questo primo esempio avremo due processi che nonostante rispettino la condizione di Liu Layland non sono schedulabili e avremo una deadline miss

Attività	A	B
Computation Time	3	4
Periodo	6 ms	9 ms

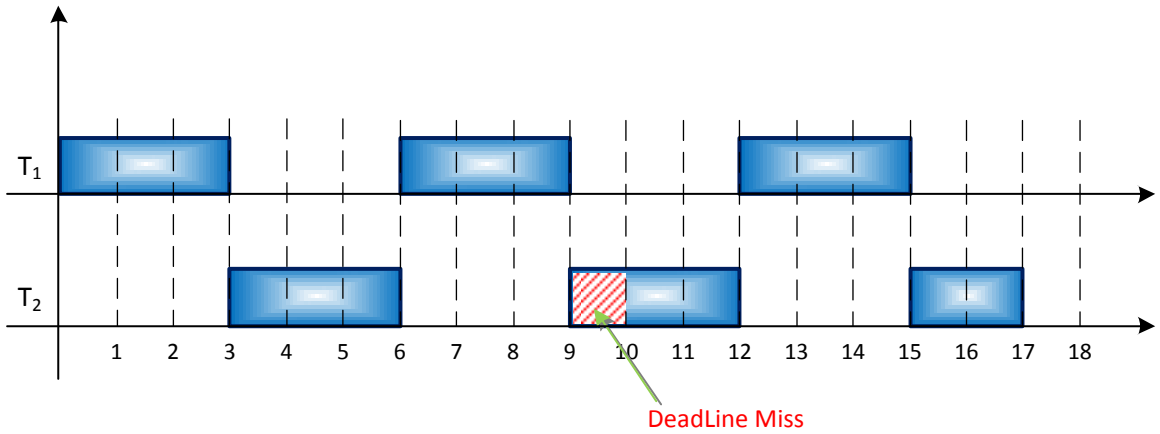


Figura 12. Esempio di Schedulazione Prioritaria non Schedulabile

2. In questo secondo esempio abbiamo un caso limite dove se aumentiamo C_1 o C_2 di un valore ε perdiamo la schedulabilità. Si parla di PIENA UTILIZZAZIONE DEL PROCESSORE.

Attività	A	B
Computation Time	2	4
Periodo	4 ms	10 ms

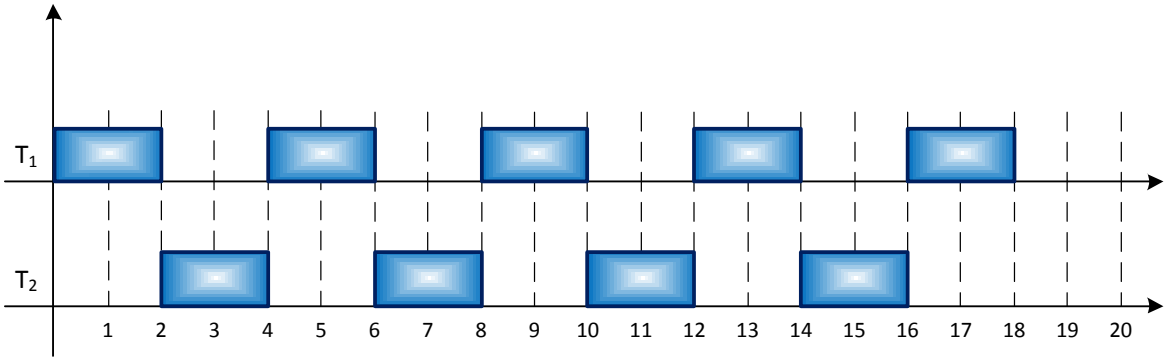


Figura 13. Esempio di Schedulazione Prioritaria Schedulabile

3. Anche in questo terzo esempio abbiamo un caso limite di piena utilizzazione del processore, l'utilizzazione è pari a 1 e la schedulazione rimane fattibile.

Attività	A	B
Computation Time	2	4
Periodo	4 ms	8 ms

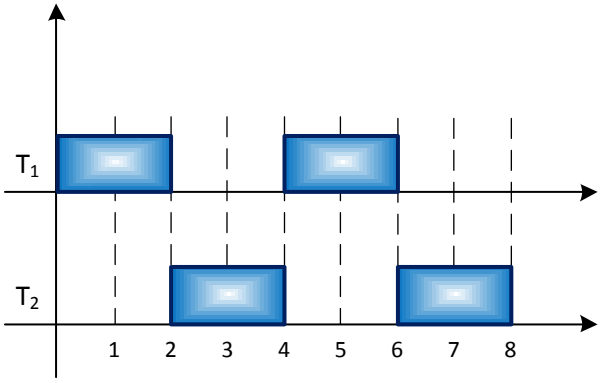


Figura 14. Esempio di Schedulazione Prioritaria con piena utilizzazione

L'idea base per individuare una **condizione sufficiente di schedulabilità** consiste nell'individuare per l'insieme dei Task considerati l'UpperBound minore, il LEAST-UPPERBOUND (U_{LUB}) dove:

$$U_{LUB} = \min[U_{Ub}(\tau)]$$

Una **Condizione sufficiente ma non necessaria** di schedulabilità è quindi che:

$$U_p \leq U_{LUB}$$

Per il calcolo dobbiamo porci in una condizione in cui si utilizzi al massimo il processore.

Procedendo nei calcoli si ricava che il valore LEAST UPPERBOUND nel caso di Rate Monotonic:

- con due Task è: $U_{LUB}{}^{RM}(2) = 2\sqrt{2} - 2 \cong 0.83$
- con *n* Task è: $U_{LUB}{}^{RM}(n) = n\left(2^{1/n} - 1\right)$
- con *n* Task dove *n* tende all'infinito: $\lim_{n\to\infty} U_{LUB}{}^{RM}(n) = \ln 2 \cong 0.69$

Graficando quindi si ottiene:

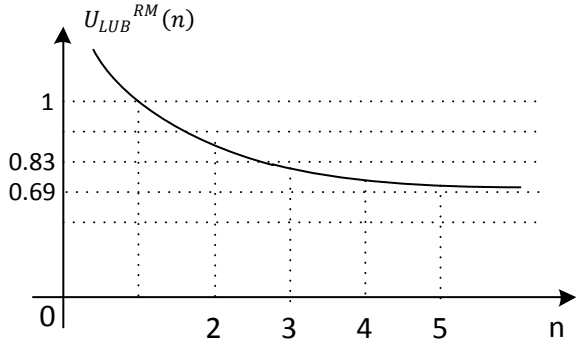


Figura 15. Grafico di U_{LUB} in funzione di n.

ANALIZZANDO IL FATTORE DI UTILIZZAZIONE per un insieme di *n* task possiamo quindi dire che:

- se $U_p > 1$: allora l'insieme dei Task non è schedulabile
- se $U_p < U_{LUB}{}^{RM}$: allora l'insieme dei Task è sicuramente schedulabile
- se $1 < U_p < U_{LUB}{}^{RM}$: non possiamo dire niente

Bini e Buttazzo (2000) sono riusciti, partendo dall'idea di Liu & Layland, a restringere la zona di indecisione, quindi se:

$$\prod_{i=1}^{n-1}(U_{i+1}) \leq 2$$

allora **L'INSIEME DI TASK È SICURAMENTE SCHEDULABILE.**

Dal seguente grafico possiamo facilmente notare come nel caso di due processi con fattore di utilizzazione U_1 e U_2 la zona di indecisione dell'algoritmo di Bini sia molto più ristretta rispetto a quella di Liu e Layland.

Possiamo inoltre dire che **l'Algoritmo di schedulazione RATE MONOTONIC è OTTIMO nel senso della schedulabilità** in quanto *se un insieme di Task è schedulabile con un algoritmo con priorità statica, allora lo sarà certamente anche con il Rate Monotonic, in altre parole se un insieme di Task non è schedulabile con l'Algoritmo Rate Monotonic non lo sarà con nessun altro algoritmo a priorità statica.*

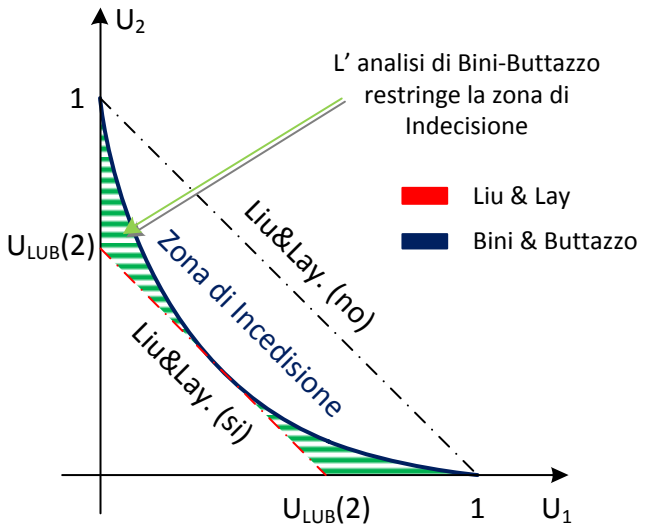


Figura 16. Zona di Indecisione dell'Algoritmo di Bini e di Liu & Layland.

1.6.2. Algoritmo Dinamico EDF

Con questo algoritmo la priorità viene assegnata in modo inversamente proporzionale alla deadline assoluta.

$$p_i \propto \frac{1}{d_i}$$

L'idea di base di questo algoritmo è l'assegnazione dinamica della priorità al Task più affamato, cioè al task la cui "Deadline assoluta d_i è più imminente".

Anche in questo caso se $U_p > 1$ l'insieme di task non è schedulabile.

Considerando il LEAST-UPPERBOUND che nel caso dell'EDF vale 1 si ottiene che **un task è schedulabile se e solo se $U_p \leq 1$.**

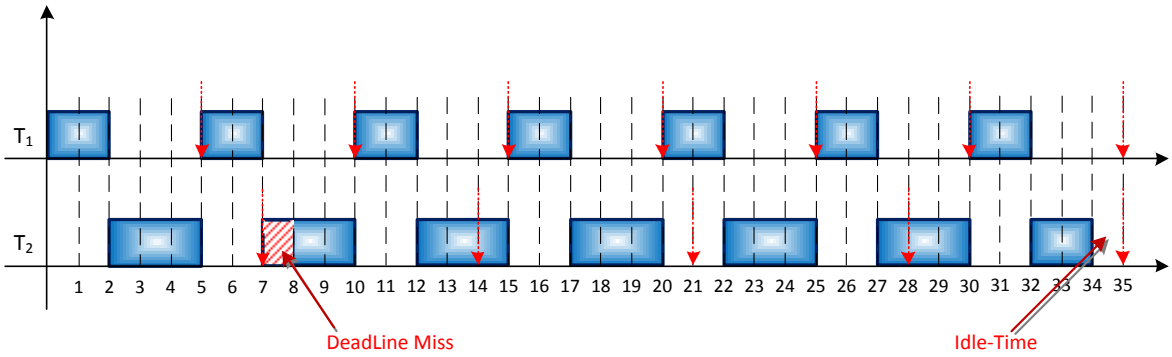
$$U_{LUB}{}^{RM}(n) = 1 \qquad \Longrightarrow \qquad U_p \leq 1$$

Se nell'algoritmo EDF viene attivato un task con deadline assoluta pari a quella del processo in esecuzione non eseguo preemption; questa è comunque una decisone del tutto arbitraria.

Vediamo un esempio di come viene schedulato un insieme di task con l'algoritmo Rate Monotonic e con l'algoritmo EDF, di questi però risulterà schedulabile solo quello con algoritmo EDF.

Attività	A	B
Computation Time	2	4
Periodo	5 ms	7 ms

Rate Monotonic



E.D.F.

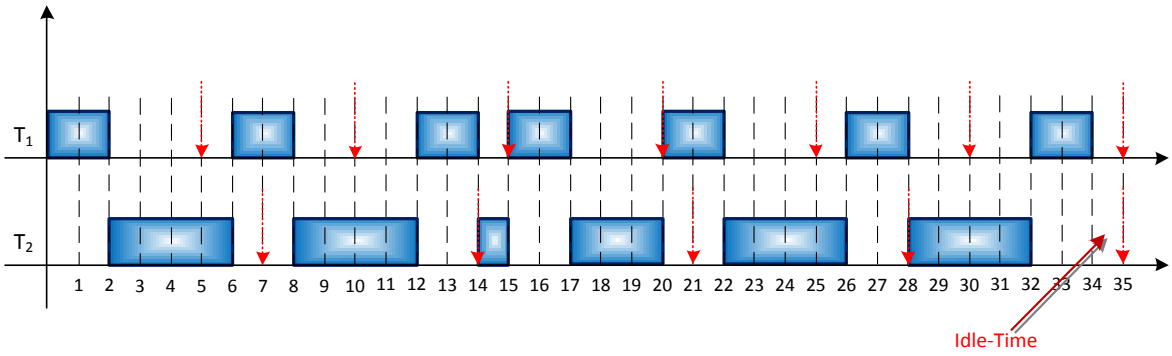


Figura 17. Confronto di Schedulazione Rate Monotonic e EDF.

1.7. Gestione Task Aperiodici con Algoritmi a Priorità Statica

Quando si sviluppano applicazioni su sistemi operativi, oltre ai task periodici, dobbiamo considerare la presenza di *attività aperiodiche* da eseguire in corrispondenza di particolari eventi esterni: i task aperiodici sono tipicamente routine di risposta a richieste d'interruzione inviate da parte di schede dedicate.

Dopo che avremo descritto le possibili soluzioni per la schedulazione dei task aperiodici, dovremo trovare dei criteri che garantiscano la schedulabilità in sistemi dove sono presenti anche dei task periodici. La schedulabilità di un insieme di task è fattibile quando ciascuno di essi termina prima della relativa deadline.

Vediamo ora cosa succede se consideriamo due task periodici schedulati con RM dove la schedulazione è fattibile, e vi andiamo ad inserire un'attività aperiodica della durata di 3 unità di tempo.

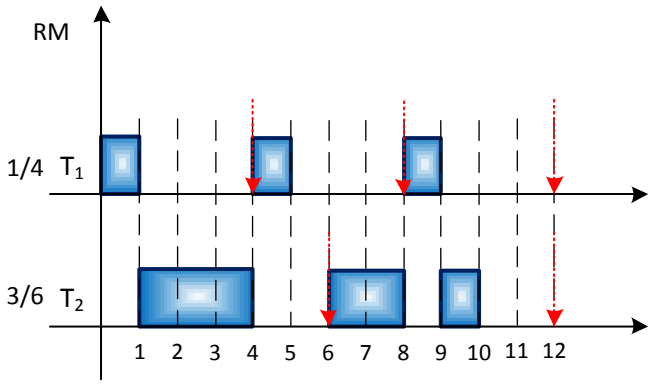


Figura 18. Si tratta di 2 Task Schedulati RM dove inseriremo un'attività Aperiodica.

Possiamo procedere in due modi:

- **servizio immediato**: dove tratteremo l'attività aperiodica come se fosse un task a priorità massima, questa è una soluzione sembra ragionevole nel caso in cui l'attività debba essere collegata ad un'interruzione esterna e debba essere servita con la maggiore celerità possibile. Il task aperiodico fa preemption su uno qualunque dei processi periodici in quando ha una priorità maggiore rispetto agli altri.

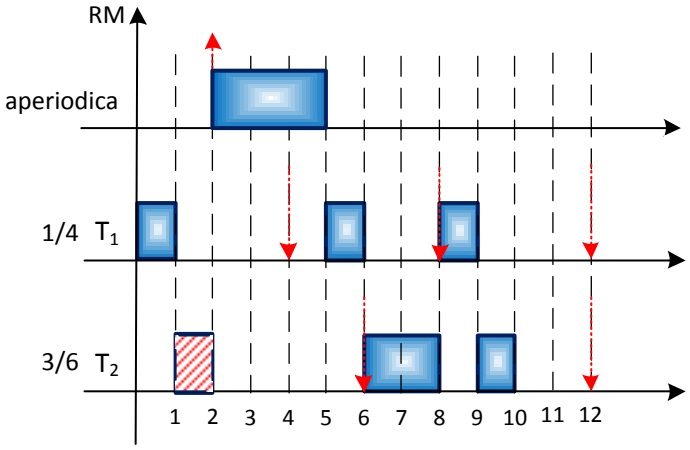


Figura 19. Ai 2 Task precedenti abbiamo inserito un'attività Aperiodica servita immediatamente al suo arrivo.

Come possiamo notare la schedulazione non è più fattibile in quanto il secondo task supera la sua deadline già nel primo periodo di esecuzione.
Questa soluzione quindi tende a privilegiare i task aperiodici nei confronti di quelli periodici, non è quindi una soluzione ragionevole in quando in un sistema real-time i task periodici e aperiodici hanno la stessa importanza.

- **schedulazione in BackGround:** in questa metodologia il sistema operativo serve le attività aperiodiche nei momenti in cui questo è libero dalla schedulazione dei task periodici, in questo caso quindi vengono privilegiate attività periodiche limitandoci ad eseguire le attività aperiodiche solo nei momenti di inattività.

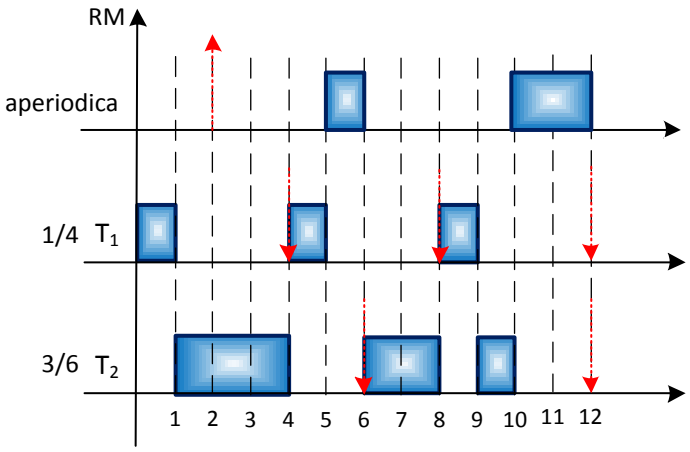


Figura 20. Ai 2 Task Periodici inseriamo un'attività Aperiodica servita in BackGround.

Come possiamo notare questa volta la schedulazione è fattibile ma degrada notevolmente il tempo di risposta del task aperiodico passando da 3 unità di tempo a 10 unità di tempo.

Il problema in questa soluzione deriva da come è distribuita l'inattività nell'iperperiodo (l'idletime nell'iperperiodo).

Calcoliamo il fattore di utilizzazione dei task periodici: $U = \frac{1}{4} + \frac{3}{6} = \frac{3}{4}$

Il Numero di Slot Temporali di Inattività nell'Iperperiodo (IDLETIME) sarà quindi:

$$IDLETIME = (1 - U) \cdot H = 3$$

Nel caso di schedulazione RM la distribuzione dell'idletime nell'Iperperiodo viene spalmata principalmente sulla parte finale. Ciò determina, nel caso di iperperiodo grande, di un tempo elevato di risposta al task aperiodico.

La **SOLUZIONE** sembra quindi essere quella di *redistribuire l'**idletime** nella fase di schedulazione dei processi e ciò può essere fatto inserendo un task apposito **(server dedicato)** per la gestione delle richieste aperiodiche.* Questo task avrà un proprio periodo e priorità e verrà trattato come un normale task; e durante la propria esecuzione si dedicherà alla gestione delle richieste aperiodiche che arrivano al sistema.

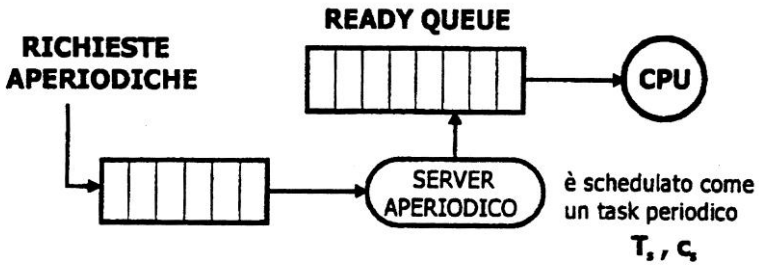


Figura 21. Schema di Funzionamento del Server dedicato ai Task Aperiodici.

Come si vede dal modello illustrato le richieste aperiodiche non arrivano direttamente al sistema, queste prima vengono inserite in una coda a loro dedicata e da questa verranno prelevate ad opera di un task server con parametri T_S , C_S .

La strategia di estrazione del task server è completamente indipendente da quella utilizzata nella schedulazione globale dei processi, e potrà quindi adottare una soluzione qualunque tra le soluzioni possibili (FIFO, prioritaria, Shortest Job Service, ecc.).

Il Task Server quindi ad ogni periodo si risveglia e controlla se nella sua coda sono presenti delle richieste aperiodiche; se la coda è vuota il server si riaddormenta, se ci sono delle richieste pendenti allora le preleva e le manda in esecuzione per un tempo massimo C_S per ogni periodo a lui assegnato T_S; se il tempo assegnato non è sufficiente all'esecuzione della richiesta aperiodica, questa verrà completata al successivo risveglio del task server.

Il task server quindi determina l'aumento del fattore di utilizzazione della quantità $\frac{C_S}{T_S}$.

Sorvoleremo l'analisi di queste tecniche di schedulazione.

1.8. Gestione Task Aperiodici con Algoritmi a Priorità Dinamica

Nel sotto capitolo precedente abbiamo affrontato il problema della schedulazione dei task aperiodici cercando di considerarli come se fossero task periodici a priorità fissa utilizzando l'algoritmo rate monotonic e cercando di minimizzare i tempi di risposta ai task aperiodici.

Ricordiamo che l'utilizzo di algoritmi dinamici permette di raggiungere la piena utilizzazione del processo.

Descriveremo ora alcune tecniche di gestione dei servizi aperiodici che utilizzano un algoritmo di schedulazione a priorità dinamica, nel nostro caso l'EDF, come prima analisi vediamo se le tecniche precedentemente viste sono utilizzabili nel caso di algoritmi a priorità dinamica:

- **Background:** questa tecnica come visto prevede la schedulazione delle richieste aperiodiche negli slot in cui il processore è idle, possiamo quindi estenderlo facilmente in quanto gli istanti di inattività del processore non dipendono dall'uso di EDF.
- **Polling Server:** anche questa tecnica può essere estesa perché il comportamento del server che si dedica alle richieste aperiodiche, nel peggiore dei casi è quello di un task periodico che quindi può essere schedulato con una certa deadline.
- **Deferrable Server:** l'estensione di questa tecnica è più complicata in quanto il

DS interagisce con il sistema offrendo un carico maggiore di U_S, ciò determina un problema in quanto la condizione di garanzia sotto EDF ($U_P + U_S \leq 1$) non è più sufficiente per la schedulabilità.

- **Sporadic Server:** questa tecnica va ancora bene perché il task per la gestione delle richieste aperiodiche si comporta come un task periodico anche se occorre modificare le regole utilizzate per il controllo e l'aggiornamento della capacità.

1.8.1. Constant Bandwidth Server (CBS)

In questo algoritmo viene definito il periodo T_S e la capacità Q_S, questi rappresentano il massimo budget che una richiesta aperiodica può consumare in un periodo T_S.

Il rapporto $\frac{Q_S}{T_S}$ rappresenta la Banda a disposizione del Server.

Si definisce:

- **C_S :** il budget attuale al tempo t (inizializzato a 0)
- **d_S :** la deadline attuale assegnata al server dopo l'arrivo di una richiesta (inizializzata a 0)

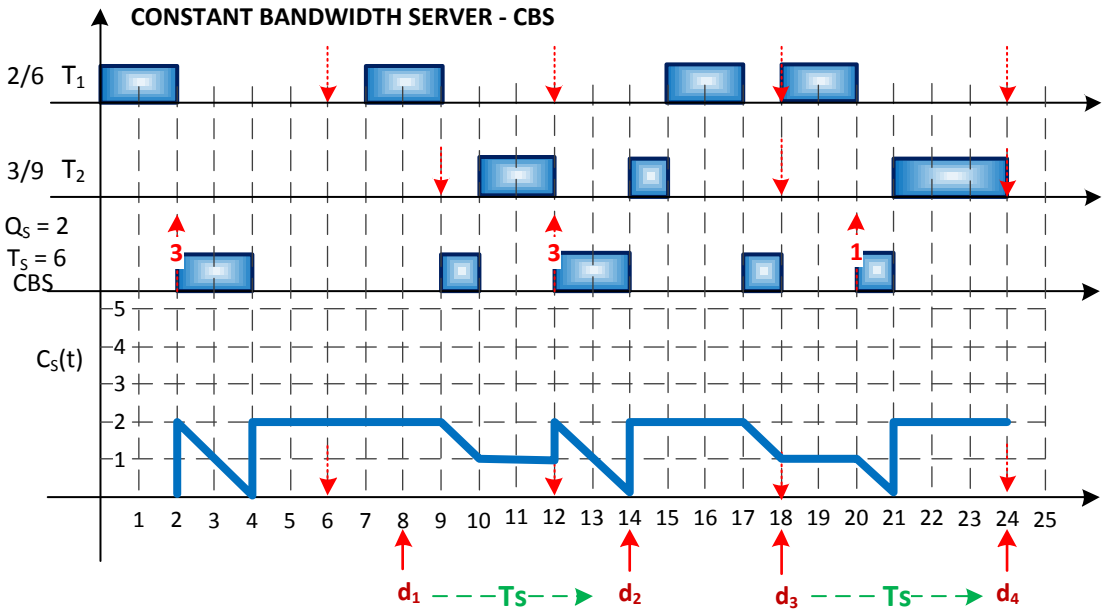
Ad ogni richiesta aperiodica servita viene assegnato un budget Q_S e una deadline relativa iniziale pari a T_S.

- *se la richiesta termina entro il budget assegnato* si passa a servire la richiesta successiva
- *altrimenti* si riassegna un nuovo budget Q_S spostando la deadline in avanti di un periodo T_S

Vediamo nello specifico l'algoritmo di questa tecnica di schedulazione:

1. Se una nuova richiesta arriva quando la richiesta attuale è ancora attiva, la nuova richiesta è accodata in una coda del server
2. Se una nuova richiesta arriva al tempo t e il server è libero:
   - Se $C_s \leq (t - d_S)U_S$ riciclo i valori attuali
   - altrimenti ricarico al massimo il budget Q_S e sposto la deadline $d_S = t + T_S$
3. Quando una richiesta viene ultimata prelevo una nuova richiesta e la schedulo con i valori attuali
4. Quando il budget si esaurisce (C_S=0) viene ricaricato al massimo e la deadline incrementata di T_S:
   - $C_S = Q_S$
   - $d_S = d_S + T_S$

Vediamo un esempio dell'utilizzo dell'algoritmo:



t	Evento	Azione	
2	Arrivo richiesta j1 – CBS idle	Cs = Qs = 2	ds = t + Ts = 8
4	Capacità esaurita	Cs = 2	ds = ds + Ts = 14
12	Arrivo richiesta j2 – CBS idle / No riciclo	Cs = Qs = 2	ds = t + Ts = 18
14	Capacità esaurita	Cs = 2	ds = ds + Ts = 24
20	Arrivo richiesta j3 – CBS idle / Si riciclo	Cs = 1	ds = 24
21	Capacità esaurita		

Figura 22. Esempio di utilizzo dell'Algoritmo Constant Bandwidth Server (CBS).

Questo metodo di schedulazione è stato trattato in quanto è quello utilizzato nella schedulazione DEAD_LINE del kernel modificato di Linux su cui abbiamo testato la nostra libreria.

1.9. Cambi di Modo nei Task

Affrontiamo ora il trattamento di una categoria di applicazioni che assumono un comportamento multi-mode. In questi sistemi ogni modo produce un comportamento differente, caratterizzato da un insieme di funzionalità svolte dai differenti insiemi di task. Un tipico esempio di comportamento multi-mode è quello di un sistema di controllo di un aereo, dove potremo distinguere la fase di *atterraggio, decollo, crociera*. Ognuno di questi modi avrà un obiettivo generale differente. Il modo di funzionamento corrente del veicolo dipenderà dalla particolare fase che stiamo eseguendo.

La suddivisione in modi di funzionamento può essere applicata in moltissimi casi, se analizziamo bene, in un qualunque sistema potremmo individuare ad esempio uno o più dei seguenti modi di funzionamento:

- **Inizializzazione:** i sistemi real-time hanno in genere tutti una fase di startup dove i dispositivi hardware e software devono essere inizializzati prima che siano attivi;
- **Check:** il sistema di Test prova se i differenti dispositivi sotto controllo stanno funzionando correttamente oppure no;
- **Emergenza:** il sistema deve funzionare con un numero ridotto di risorse, per esempio, a causa di un'interruzione di corrente;
- **Allarme:** il sistema deve essere portato su un piano di controllo particolare ad esempio a causa di un incidente o una situazione di rischio;
- **Salvataggio:** alcuni sistemi quali gli stimolatore cardiaci o i registratori automatici di temperatura hanno un modo di operazione speciale dove memorizzano i dati storici ottenuti durante il loro funzionamento normale;
- **Low Power:** in questo caso il sistema deve ottimizzare l'alimentazione elettrica minimizzando il numero delle attività correnti o la loro frequenza per ridurre l'assorbimento corrente di energia;
- **Alta Affidabilità:** quando vogliamo assicurarci la correttezza del funzionamento di un sistema potremmo ad esempio utilizzare delle tecniche di ridondanza;
- **Recupero da Errore:** in questo caso vorremo effettuare una re-inizializzazione dei task che hanno generato l'errore e avviare un sistema diagnostico di recupero.

Un cambio di modo inizia ogni volta in cui il sistema rileva un cambiamento nell'ambiente o nella condizione interna in cui si viene a trovare il sistema stesso. Per cambiare il modo di funzionamento corrente è necessario cancellare alcuni task e lanciarne altri. Si introduce quindi una certa fase transitoria in cui l'insieme dei task in esecuzione può comprendere sia task del vecchio che del nuovo modo, ciò può determinare un sovraccarico temporale. Dobbiamo quindi estendere il concetto di fattibilità a questi sistemi in modo da garantire che nessuna deadline sia mancata

durante la transizione tra i due modi.

1.9.1. Definizioni e Modello di Sistema

Si definisce **Modo di Funzionamento** il comportamento assunto dal sistema in un determinato istante; questo sarà descritto da un serie di funzionalità e dalle sue relative limitazioni temporali per la schedulazione, un modo è quindi descritto da un insieme di Task. Quindi quando sarà attivo un determinato modo dovranno essere attivi anche i relativi task (1).

A seguito di una **richiesta di cambiamento di modo (MCR)** si innesca una *transizione* del sistema da un modo di origine (o vecchio modo) a un modo di destinazione (o nuovo modo). Questo evento può essere scatenato da un qualunque task che possiede la necessaria conoscenza dello stato interno o esterno del sistema che può produrre la transizione, ad esempio un Task che controlla un sensore di allarme può decidere di inviare un MCR per "allarmare" il gestore dei cambi di modo del verificarsi di un evento.

Una richiesta di cambio di modo può essere inviata solo quando il sistema si trova stabilmente in un determinato modo di funzionamento, non potrà essere inviata se è già in atto una transizione tra due modi. Il sistema deve evitare MCR simultanei o fornire un meccanismo che si occupa di questo evento, ad esempio, usando una gerarchia dei modi di funzionamento.

Per la nostra analisi considereremo che in ogni modo di funzionamento un Task T_i venga descritto dalla tupla composta da C_i, T_i, D_i, P_i (Computation Time, Period, Deadline, Priority). Se abbiamo quindi m modi di funzionamento da M_1a M_m potremo descrivere il Task T_i come un insieme di tuple:

$$\tau_i = \begin{Bmatrix} \tau_i^{M_1} = \left(C_i^{M_1}, T_i^{M_1}, D_i^{M_1}, P_i^{M_1}\right), \\ \tau_i^{M_2} = \left(C_i^{M_2}, T_i^{M_2}, D_i^{M_2}, P_i^{M_2}\right), \dots, \\ \tau_i^{M_m} = \left(C_i^{M_m}, T_i^{M_m}, D_i^{M_m}, P_i^{M_m}\right) \end{Bmatrix}$$

Un Task può o non può esistere in un particolare modo, se in un modo non è attivo allora avremo che $C_i^M = 0$. Alcuni studiosi inoltre tengono in considerazione che i task possono cambiare i tempi di funzionamento a seconda del modo in cui ci troviamo.

Il modello descritto sarebbe il più flessibile, ma presenta alcuni svantaggi:

1. se il task contiene del codice dipendente dal modo di funzionamento in cui si trova, questo, nel caso aggiungessimo nuovi modi di funzionamento al sistema, potrebbe dover essere sottoposto a modifiche, e ciò ne deteriorerebbe la proprietà di riusabilità. Il problema potrebbe essere risolto con le tecniche orientate agli oggetti;
2. l'analisi di schedulabilità risulta molto più complicata

Nelle nostre considerazioni e analisi assumeremo che un Task T_i sia identificato da una porzione ben definita di codice e che questo non dipenda dai modi dove ci troviamo. Quindi in una fase di transizione tra due modi M_a e M_b in cui il Task T_i à attivo avremo $C_i^{M_a} = C_i^{M_b}$. In base a quanto detto, nel caso in cui un'applicazione richieda un Task che varia il proprio comportamento in funzione del modo in uso, noi modelleremo il sistema con un Task separato per ogni tipologia di comportamento.

1.9.2. Classificazione dei Task in base alla MCR

Se abbiamo una richiesta di cambio di modo (MCR) possiamo classificare i Task T_i in base al loro comportamento durante la fase di transizione (1).

Se il **Task T_i appartiene al vecchio modo** può essere:

- *Old-Mode completed Task,* $t_{i(O)}$: Il task deve completare per intero la propria esecuzione, in questo caso parte della sua esecuzione avviene dopo che abbiamo ricevuto la MCR;
- *Old-Mode aborted Task,* $t_{i(A)}$: Il task viene abortito quando viene lanciata la MCR. Il Task abortito forniva una funzionalità che non è più necessaria nel nuovo modo, in questo caso l'aborto è fattibile senza che si perda la consistenza dei dati.

Se il **Task T_i appartine al nuovo modo** può essere:

- *New-Mode, Changed Task,* $t_{i(C)}$: Il Task T_i è attivo in entrambi i modi, ma i suoi relativi parametri (periodo e deadline) cambiano dal vecchio al nuovo modo;
- *New-Mode, unChanged Task,* $t_{i(U)}$: Il Task T_i è attivo in entrambi i modi e mantiene il medesimo comportamento temporale (periodo e deadline). Se un task viene classificato in questo modo vuol dire che esiste una corrispondente versione di questo nel vecchio modo;
- *Wholly, New Task,* $t_{i(W)}$: Il Task T_i appartiene al nuovo modo ma non era presente in quello vecchio.

Definiamo un nuovo parametro detto **offset**, questo è il ritardo che un protocollo può imporre alla prima attivazione di un task nel nuovo modo. Questo valore è un tempo di ritardo che viene calcolato a partire dall'istante in cui viene lanciata una MCR. L'applicazione di un offset ai Task che devono essere attivati nel nuovo modo ci permette di ridurre il sovraccarico prodotto dalla MCR stessa.

Il valore dell'offset dipende sia dai vecchi che dai nuovi modi e può variare in base all'utilizzazione stessa del sistema: se l'utilizzazione è bassa in entrambi i modi, allora un Task del nuovo modo può essere introdotto prima senza introdurre sovraccarichi, se invece l'utilizzazione è alta potremmo dover applicare un offset più grande per ridurne il sovraccarico. Un Task T_i è quindi caratterizzato da una matrice bidimensionale degli offset Y_i da applicarsi nelle diverse transizioni dei cambi di modo.

Le righe di Y_i rappresentano i modi di origine e le colonne i modi di destinazione. Quindi nel caso di un cambio di modo da M_a a M_b indicheremo il relativo offset come $Y_i^{M_a,M_b}$.

Per un sistema con i cambi di modo, la **deadline D_i** che applichiamo ad un nuovo Task indica il tempo massimo concesso per il completamente della prima attivazione del task rispetto al tempo in cui l'MCR è stata lanciata. Considerando sia l'offset che il tempo di reazione R_i del Task dovrà valere la seguente relazione:

$$R_i + Y_i \leq D_i$$

Definiamo **Latenza del Cambio di Modo** l'intervallo di tempo fra la MCR e la conclusione della transizione. Dobbiamo definire cosa si intende per conclusione della transizione in quanto possiamo avere varie interpretazioni; ad esempio, può esistere un task del vecchio-modo con un grande periodo di esecuzione che è in funzione durante la transizione e che viene completato molto tempo dopo la MCR, mentre il resto dei task del nuovo-modo deve essere completato ben prima. In questo caso, misurare lo stato latente del cambiamento di modo in funzione del task con periodo più lungo, se ad esempio abbiamo avuto un cambio di modo a seguito di un'interruzione di alimentatore, può non avere più significato rispetto agli altri Task.

In Pedro(1999) la Latenza comprende l'esecuzione di tutti i task del vecchio modo più la prima attivazione di tutti i task del nuovo modo.

1.9.3. Requisiti richiesti nel cambio di modo

Definiamo ora i requisiti che saranno considerati come gli obiettivi da realizzare durante la transizione del cambio di modo. Questi requisiti ci permetteranno di valutare e classificare i vari protocolli.

- **Schedulabilità**: prima, dopo e durante la transizione il sistema deve rimanere schedulabile. Il sovraccarico prodotto dalla MCR non dovrebbe indurre a mancare nessuna deadline.
- **Periodicità**: i task che sono presenti in entrambi i modi devono continuare ad eseguire periodicamente come se non fosse accaduto niente (ad esempio un task campionatore attivo in entrambi i modi deve continuare regolarmente il proprio compito)
- **Prontezza**: la transizione tra un modo e l'altro dovrebbe avvenire nel più breve intervallo di tempo possibile.
- **Consistenza delle Variabili**: le risorse devono rimanere consistenti anche durante il cambio di modo

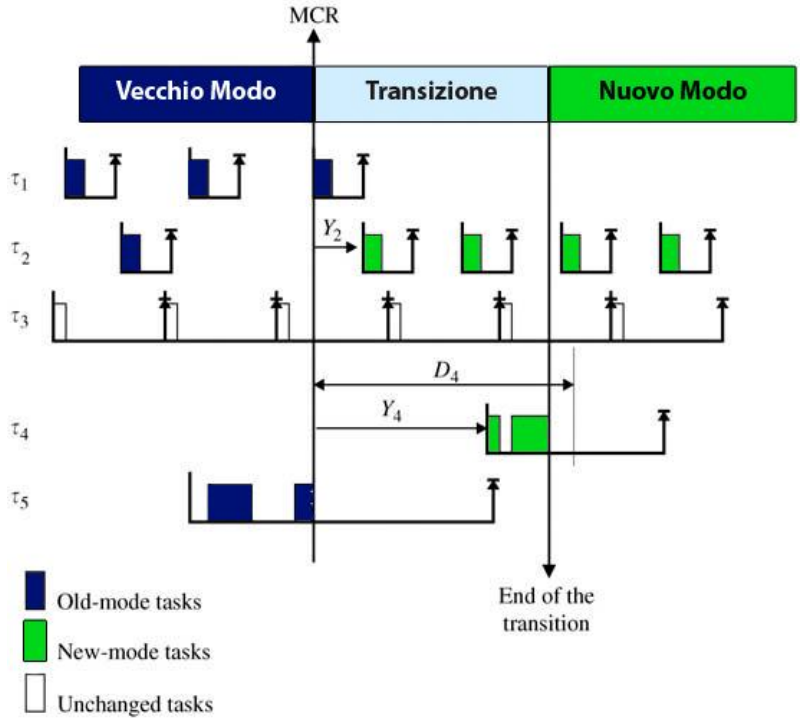


Figura 23. Definizioni relative ai Cambi di modo

La figura illustra graficamente i differenti tipi di mansioni e i loro comportamenti su richiesta del cambiamento di modo. Il Task T1 è un task del vecchio-modo che è attivo solo nel vecchio modo. La relativa ultima attivazione coincide con il tempo in cui arriva la MCR. Il task T2 è attivo in entrambi i modi ma nel cambio di modo modifica i parametri caratteristici del modello di attivazione, aumentando la relativa frequenza nel nuovo modo. La prima attivazione di T2 è fatta ritardare da un offset Y2, riferito a MCR. Il T3 è un task che rimane immutato quindi la sua attivazione è indipendente dal cambiamento di modo in atto. T4 è un task interamente nuovo, introdotto dopo un offset Y4 e con una deadline di cambio di modo D4. Infine T5 è un task abortito del vecchio-modo. La Latenza del cambio di modo è stata considerata come l'intervallo di tempo fra il MCR ed il completamento della prima attivazione di tutte le mansioni di nuovo-modo.

Cap. 2 - La libreria pthread

2.1. Introduzione

Il codice di programmazione viene in genere scritto in modo serializzato, il codice viene cioè eseguito sequenzialmente, ogni riga di codice viene eseguita una dopo l'altra senza valutare se è possibile incrementare le prestazioni utilizzando una sorta di programmazione parallela che sfrutta i Thread. Con l'aumento delle diffusione delle macchine con symmetric multiprocessing (dovuto alla nascita dei processori multi-core), l'utilizzo dei Thread è diventato molto più consueto.

2.1.1. Cos'è un Thread ?

Consideriamo la seguente analogia, associamo il concetto di processo a quello di un ago da cucire e quello dei thread ad un filo: se per terminare un lavoro utilizziamo due aghi ma un solo filo impiegheremo un tempo X, se dividiamo il filo in due e utilizziamo due aghi e due fili possiamo pensare che impiegheremo la metà del tempo precedente $X/2$.

Come già detto, un programma quando viene elaborato diventa un processo memorizzato in qualche spazio di memoria del computer e inizia la propria esecuzione. Un processo può essere eseguito da un processore o un insieme di processori. Un processo in memoria dovrà quindi contenere tutte le informazioni che lo riguardano come il puntatore alla posizione corrente del programma (IP), i registri di sistema, le variabili, gli handle dei file, ecc.

*Un **Thread** non è altro che una sequenza di tali istruzioni che può essere eseguita indipendentemente dagli altri Thread*.

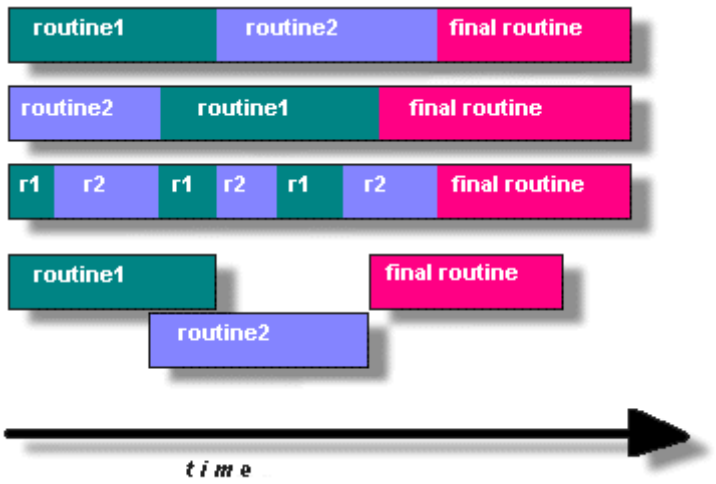


Figura 24. Thread Indipendenti che possono essere eseguiti in modo parallelo.

Dalla figura seguente possiamo dedurre che i thread sono all'interno dello spazio di indirizzamento del processo, quindi gran parte delle informazioni presenti nel descrittore del processo potranno essere condivise con il Thread stesso.

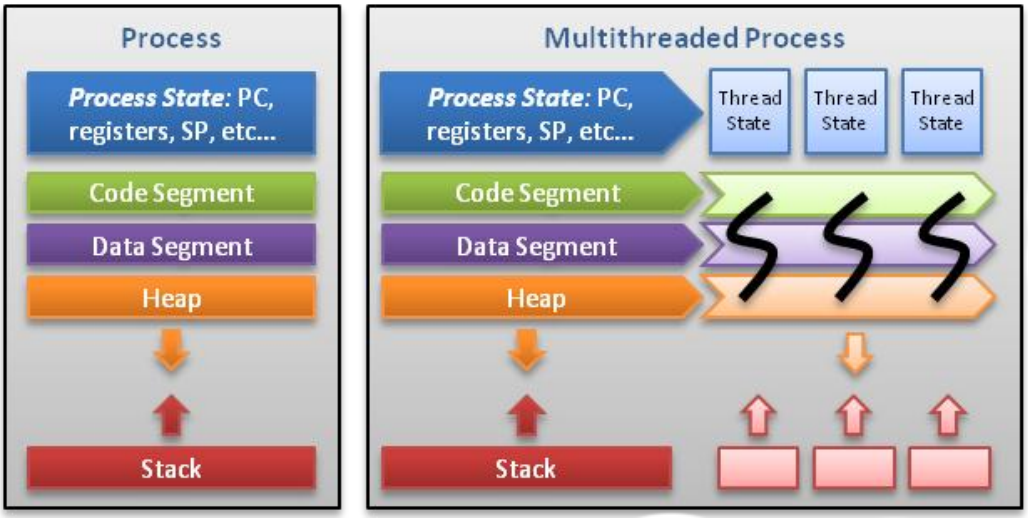


Figura 25. Risorse condivise in un Thread.

Alcune informazioni non possono essere replicate, come ad esempio lo Stack, i registri e i dati specifici del thread. Questo determina che il sistema operativo dovrà fornire dei meccanismi specifici per la gestione dei programmi in multithread principalmente per riuscire a mantenere i dati consistenti, è stato infatti necessario sviluppare metodi per la gestione della concorrenza sulla memoria che è quindi comune.

Fortunatamente tutti i moderni sistemi operativi (Linux, BSD, Mac OS X, Windows, Solaris, ecc.) supportano i thread, chiaramente ogni sistema implementerà il supporto ai multithread in modo diverso.

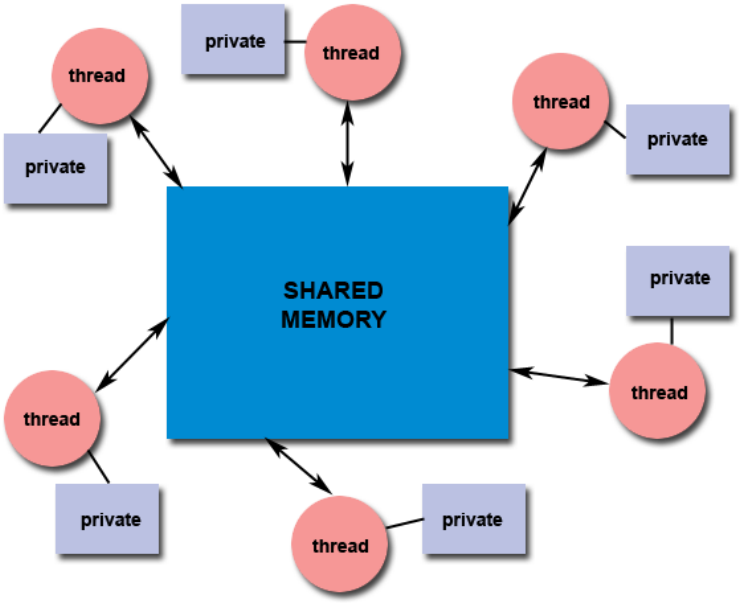


Figura 26. Modello a Memoria Comune.

I thread possono quindi essere eseguiti contemporaneamente esattamente come avviene per i processi, ma allora perché preferire il multithreading a più processi indipendenti ? I Thread come detto condividono lo stesso spazio di memoria, Thread indipendenti possono quindi accedere alle stesse variabili in memoria. Infatti tutti i thread contenuti nel programma possono leggere e scrivere le stesse variabili dichiarate globalmente. Questo è un problema che invece ritroviamo utilizzando la funzione fork(), questa crea due processi distinti ma lascia aperto il problema della comunicazione tra di essi. In questo caso dovremo adottare un protocollo di comunicazione tra processi, ma ritroviamo due inconvenienti: il kernel impone un overhead in più abbassando le prestazioni e spesso l'utilizzo del protocollo di comunicazione accresce notevolmente la complessità del programma.

L'utilizzo dei Thread al posto dei Processi determina quindi un notevole risparmio di tempo di CPU, e rende la creazione dei Thread da decine a centinaia di volte più veloce rispetto alla creazione di un nuovo processo. Possiamo quindi utilizzare i thread in qualsiasi momento ne abbiamo bisogno all'interno del programma.

Tutto ciò è dimostrato anche dalle seguenti tabelle dove vengono messi a confronto i risultati di temporizzazione per la *fork()* e il sottoprogramma *pthread_create()*. Per il calcolo dei tempi sono stati mandati in esecuzione 50000 processi/thread su macchine differenti e il tempo indicato è in secondi (2):

Piattaforma	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.8 GHz Xeon 5660 (12cpus/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Proprio come i processi, i Thread trarranno vantaggio dalla presenza di più CPU, se il software è progettato per essere utilizzato su macchine multiprocessore le prestazioni saliranno in modo abbastanza lineare relativamente al numero dei processori presenti nel sistema. Se scriviamo un programma che usa intensamente la CPU, sarà molto importante riuscire ad utilizzare i thread al suo interno.

Il seguente esempio (2) mostra la differenza di prestazioni che abbiamo appunto nella semplice comunicazione tra processi e tra thread:

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

2.1.2. Cosa sono i PThread ?

Storicamente i produttori di sistemi operativi hanno implementato le loro versioni proprietarie per i thread; queste implementazioni differivano in modo sostanziale l'una dall'altra, rendendo ai programmatori difficile lo sviluppo di applicazioni che le utilizzassero. Al fine di utilizzare al meglio le funzionalità offerte dai Thread, è stata richiesta un'interfaccia di programmazione standard.

Per i sistemi UNIX, questa interfaccia è stata specificata dalla IEEE POSIX 1003.1c standard (1995). Le implementazioni che aderiscono a questo standard vengono dette PosixThreads o Pthreads. La maggior parte dei fornitori di hardware offre quindi PThreads in aggiunta alle loro API proprietarie. Lo standard Posix ha continuato ad evolversi e a sottoporsi a revisioni includendo le specifiche PThreads.

Alcuni link utili che riguardano lo standard sono:

- standards.ieee.org/findstds/standard/1003.1-2008.html
- www.opengroup.org/austin/papers/posix_faq.html
- www.unix.org/version3/ieee_std.html

I Pthreads sono definiti come un insieme di tipi di dati e di chiamate di procedure scritti in linguaggio di programmazione C e sono implementati per mezzo di un file di intestazione da includere *"pthread.h"* e di una libreria che in alcuni casi può far parte di un'altra libreria come la *"libc"*.

2.2. Dalla Creazione alla Terminazione di un Thread

Un thread è identificato da un oggetto di tipo *pthread_t*, è il così detto "thread id", l'identificatore (il gestore) del thread.

```
pthread_t my_thread;
```

Vediamo un esempio di creazione e terminazione di un Thread, poi analizzeremo le singole istruzioni:

```
/* Include */
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
/* Corpo del thread */
void *PrintHello(void *num) {
  printf("\n%d: Hello World!\n", num);
  pthread_exit(NULL)
}

/* Programma */
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0; t<NUM_THREADS; t++){
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t],
                        NULL,
                        PrintHello,
                        (void *)t);
    if (rc){
      printf("ERROR;
          return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++)
  {
    rc = pthread_join(thread[t], (void **)&status);
    if (rc) {
      printf("ERROR;
          return code from pthread_join() is %d\n", rc);
      exit(-1);
    }
    printf("Completed join
        with thread %d status= %d\n",t, status);
  }
  pthread_exit(NULL);
}
```

2.2.1. Creazione del Thread

Dopo che il my_thread è stato dichiarato, chiamiamo la funzione phtread_create per creare un vero e proprio thread. In genere si pone la funzione di creazione all'interno di un if() ciò in quanto la funzione ritorna 0 in caso di successo un valore diverso da zero in caso di insuccesso.

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

- *pthread_t *thread* : rappresenta l'oggetto thread attuale e contiene il thread id
- *pthread_attr_t *attr* : sono gli attributi che vogliamo specificare del thread
- *void *(*start_routine)(void *)* : è la funzione che il thread esegue
- *void *arg* : sono gli argomenti passati alla funzione che esegue il thread

Gli attributi che possiamo passare a un thread vengono impostati in una variabile di tipo *pthread_attr_t* che viene inizializzata per mezzo della funzione *pthread_attr_init*, i vari settaggi avverranno poi per mezzo di specifiche funzioni di impostazione[6]. Se nella chiamata della creazione del thread si pone tale parametro come NULL, verranno utilizzati i parametri di default.

```
pthread_attr_t my_attr;
int pthread_attr_init(pthread_attr_t *my_attr);
```

2.2.2. Terminazione del Thread

```
void pthread_exit(void *value_ptr);
```

Invocando questa funzione ritorniamo dalla funzione thread dalla quale è stato chiamato. In pratica viene chiamata la routine di cleanup del pthread (*pthread_cleanup_push()* o similare), termina la sua esecuzione salvandone lo stato in modo che possa essere recuperato dalla funzione join.

Il parametro passato infatti risulta essere un parametro di ritorno e può essere ottenuto tramite la funzione join del chiamante.

2.2.3. Attesa di terminazione

```
int pthread_join(pthread_t my_thread, void **value_ptr);
```

Sospende il processo che lo invoca finché il thread identificato da *my_thread* termina.

Se *&status != Null* salva in *status* lo stato della terminazione. Normalmente quando un thread termina, la memoria occupata dal suo stack privato e la posizione della tabella

[6] Detached o Joinable state;
Scheduling inheritance, policy, parameters, contention scope;
Stak size, adddrer, guard

dei thread non vengono rilasciate finché qualcuno non chiama la *pthread_join()* su quel thread, se quindi non chiamiamo la *pthread_join()* causiamo memory leak.

Se invochiamo la *pthread_detach()* questa ci permette di liberare le risorse senza attendere il join (se abbiamo però invocato la detach su quel thread non potremo più invocare la join).

2.3. Meccanismi di sincronizzazione

Come abbiamo precedentemente detto, i thread lavorano su ambiti di memoria comune nasce quindi la necessità di un meccanismo che gestisca la concorrenza a tali dati e risorse, in modo da definire un protocollo per l'accesso sicuro. Serve cioè un meccanismo esplicito che blocchi gli altri thread quando questi cercheranno di accedere a delle risorse che stiamo già utilizzano noi.

2.3.1. La Mutua Esclusione

La Mutua Esclusione è il meccanismo che noi abbiamo per serializzare l'accesso alle risorse condivise. Non vogliamo cioè che un thread modifichi un dato che già è in procinto di essere modificato da un altro thread. Un altro caso che vogliamo evitare è la lettura sporca che un thread può avere se è in corso l'aggiornamento dei dati da parte di un altro thread.

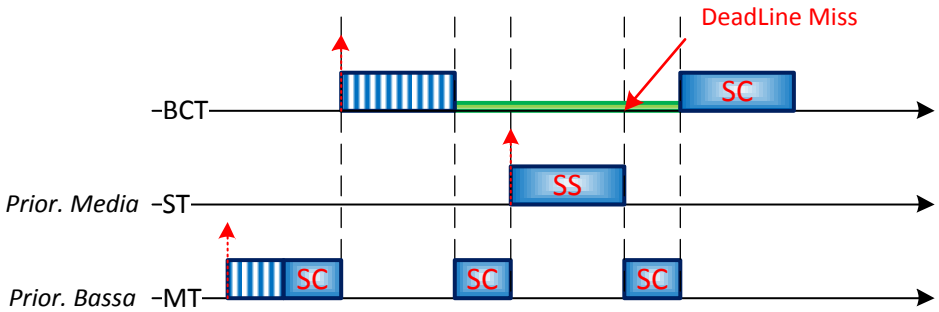
Il **mutex** è quindi un blocco che possiamo collegare a una risorsa: se un thread vuole modificare o leggere un valore da una risorsa condivisa, dovrà prima ottenerne il blocco. Una volta ottenuto il blocco possiamo fare ciò che vogliamo sulla risorsa condivisa, senza doverci preoccupare degli altri thread che vi potrebbero accedere, questi infatti dovranno attendere che noi sblocchiamo il mutex, a questo punto i thread che erano in attesa dovranno gareggiare alla conquista dell'utilizzo della risorsa e ogni thread dovrà prevedere un meccanismo che gli garantisca l'utilizzo esclusivo della risorsa.

La parte di codice compresa tra il blocco e lo sblocco del semaforo mutex viene comunemente definita **sezione critica**. Minore è il tempo speso nella sezione critica e maggiore è la concorrenza in quanto si riduce la quantità di tempo che ogni thread deve attendere per ottenere il blocco.

L'utilizzo dei mutex deve però essere consapevole in quanto può portare a diverse problematiche:

- **deadlock**: un thread può continuare a ciclare permanentemente o arrestare la propria esecuzione senza sbloccare il mutex, i thread che sono in attesa di utilizzare la risorsa condivisa rimangono quindi bloccati.
- **race condition**: si verifica quando non rispettiamo un determinato protocollo di accesso a una risorsa, ciò può determinare risultati errati che causano guasti o comportamenti incoerenti.

- **inversione di priorità**: si verifica quando un thread a priorità più alta non può eseguire perché bloccato sulla sezione critica di una risorsa condivisa con un processo a più bassa priorità che a sua volta potrebbe essere bloccato da un thread a più bassa priorità.



Come possiamo notare dal grafico abbiamo deadline miss in quanto BCT vuole ottenere l'uso della risorsa SC ma questa è bloccata da MT che ha subito preemption da ST.

Detto questo lo standard POSIX prevede l'utilizzo di un oggetto *pthread_mutex_t* che implementa i semafori mutex:

```
pthread_mutex_t my_mutex;
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

La funzione di inizializzazione del semaforo mutex richiede come primo parametro una variabile *pthread_mutex_t* che verrà inizializzata tramite i parametri specificati nel secondo parametro, per settare gli attributi di *default* del semaforo possiamo utilizzare il valore NULL come secondo parametro. In alternativa possiamo inizializzare il semaforo per mezzo di comode macro invece delle chiamate di funzione

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

L'oggetto mutex inizializzato potrà quindi essere bloccato e sbloccato tramite le seguenti funzioni:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

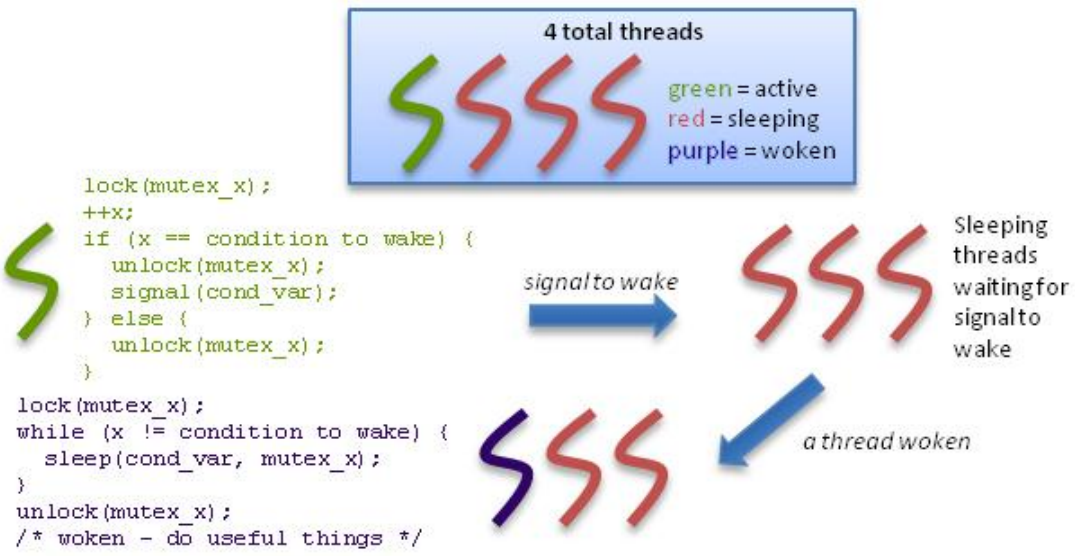
In ognuna delle chiamate è necessario specificare l'oggetto mutex, la differenza tra la *lock* e la *trylock* consiste nel fatto che la lock si blocca in attesa che avvenga lo sblocco, la trylock non si blocca, ritorna immediatamente, sarà quindi necessario controllare il valore di ritorno della chiamata in modo da determinare se il mutex è stato acquisito con successo o meno. Se non è stato acquisito il blocco sarà restituito il codice di errore EBUSY diverso da zero.

2.4. Le Variabili Condizionali

Le variabili condizionali vengono utilizzate ad esempio nel caso in cui vogliamo aspettare il verificarsi di una certa condizione su dei dati condivisi. Utilizzando i mutex dovremmo sbloccarli e ribloccarli ripetutamente controllando ogni volta il cambiamento del valore. Questo è chiaramente un brutto approccio, in quanto il thread di controllo dovrebbe rimanere sempre attivo. Potremmo pensare di utilizzare uno sleep in modo da sospendere il thread tra un controllo e l'altro ma ciò determinerebbe un thread poco reattivo.

Le variabili condizionali risolvono quindi questo problema, mettono in sleep il thread e aspettano che alcune condizioni vengano a verificarsi. Una volta che si è verificata la condizione attesa, la variabile condizionale sveglia il thread che si era bloccato.

Ad esempio potremmo pensare di aver un contatore che una volta raggiunto un certo valore sveglia un thread. Chiaramente stiamo supponendo che il thread che viene svegliato fosse in attesa sulla variabile condizionale. L'utilizzo delle variabili condizionali ci permette anche di lasciare più thread in attesa e di svegliarli tutti insieme.



Nell'esempio illustrato dovremmo però inserire l'attesa all'interno di un ciclo while() non all'interno di un semplice if() a causa di eventuali **wakeups spurie**. Non abbiamo infatti la garanzia che un thread si svegli correttamente in seguito a un segnale o a una chiamata di broadcast.

Nello standard POSIX la creazione delle variabili condizionali avviene in modo simile a quella vista in precedenza per i mutex:

```
pthread_cond_t my_cond;
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

Anche in questo caso la funzione di inizializzazione del semaforo richiede come primo parametro la variabile *pthread_cond_t* rappresentante il semaforo dichiarato, e come secondo i parametri specifici di inizializzazione, per settare gli attributi di *default* del semaforo possiamo passare il valore NULL come secondo parametro. In alternativa possiamo inizializzare il semaforo ai valori predefiniti per mezzo di comode macro anziché utilizzare la chiamata di funzione:

```
pthread_cond_t my_cond = PTHREAD_COND_INITIALIZER;
```

L'oggetto condizionale inizializzato potrà sincronizzarsi tramite le seguenti funzioni:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

La funzione *wait()* mette il thread corrente a dormire e richiede come parametro anche il mutex che "protegge" la variabile condizionale condivisa che è in attesa.

La funzione *signal()* sveglia un eventuale thread che potrebbe essere bloccato sulla variabile condizionale; la funzione *broadcast()* invece sveglia tutti i thread in attesa sulla variabile condizionale.

2.5. Barriere

Le barriere sono dei metodi per sincronizzare una serie di thread su un certo punto, tutti i thread che partecipano alla barriera aspettano fino a quando tutti i thread non hanno chiamato la funzione della barriera detta. Una barriera in pratica blocca tutti i thread che vi partecipano su un punto di sincronizzazione indicato fino a quando il thread più lento non raggiungere anche lui la chiamata della barriera.

L'oggetto Barriera nello standard POSIX viene così inizializzato:

```
pthread_barrier_t my_barrier;
int pthread_barrier_init(pthread_barrier_t *barrier,
                         pthread_barrierattr_t *barrier_attr,
                         unsigned int count);
pthread_barrier_t barrier =
                         PTHREAD_BARRIER_INITIALIZER(count);
```

Il metodo di inizializzazione è simile a quello utilizzato in precedenza per i semafori. In più c'è solo il parametro *count* dove dobbiamo specificare il numero di thread che devono sincronizzarsi.

Per sincronizzarsi l'oggetto barriera chiama la funzione:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Tale funzione non fa altro che bloccare il thread che la invoca, quando la funzione è

stata chiamata da tanti thread quanto abbiamo indicato in count, la funzione sveglia contemporaneamente tutti i thread dormienti che si erano sincronizzati.

2.6. Ulteriori funzioni utili della libreria pthread

```
pthread_self()
```

restituisce un handle del thread chiamante.

```
pthread_equal()
```

confronta l'uguaglianza tra due pthread ID

```
pthread_once()
```

può essere usato per assicurare che una funzione di inizializzazione all'interno di un thread sia eseguita solo una volta.

Ci sono molte altre funzioni utili nella libreria pthread. Per queste rimandiamo la consultazione alle pagine specifiche (2).

Cap. 3 - La libreria boost

Le Librerie C++ Boost sono una collezione di librerie open source che estendono le funzionalità del C++. Molte di esse sono licenziate sotto la Boost Software License in modo da poter essere utilizzate sia in progetti open source che closed source. Alcuni dei fondatori di Boost fanno parte del comitato standard C++ (ISO/IEC 14882) e diverse librerie Boost sono state accettate per l'incorporazione sia in C++ Technical Report 1, sia in C++0x[1].

Per assicurare efficienza e flessibilità, Boost fa un estensivo utilizzo della programmazione basata su template, e quindi sulla programmazione generica e metaprogrammazione.

Boost rappresenta una collezione di librerie, alcune delle quali sono semplici, come le utility di conversione, e altre più complesse. Ci sono delle librerie che potremmo utilizzare molto spesso: sistema di manipolazione dei file, algoritmi grafici, gestione della memoria heap, espressioni regolari, utilità di conversione, zip, ecc.

3.1. Perché scegliere Boost ?

- Boost tiene conto dei namespace. Tutti i componenti all'interno della libreria sono confezionati nello spazio dei nomi "boost"
- Boost è una libreria che è in continua crescita, continuamente quindi vengono rilasciati degli aggiornamenti
- C'è un supporto agli sviluppatori, domande relative ai componenti possono essere indirizzate alla boost users mailing group
- Boost supporta una varietà di compilatori, sistemi operativi e librerie standard
- Test di regressione, ogni aggiornamento di Boost viene fortemente testato
- Molte persone coinvolte con lo sviluppo dello Standard C++ sono coinvolti con Boost
- E' semplice da installare e aggiornare
- E' facile da configurare

3.2. Cosa abbiamo utilizzato di Boost

Elenchiamo brevemente le librerie della Boost che abbiamo utilizzato nel nostro sviluppo:

3.2.1. Bind e Function

Questi due oggetti sono sviluppati come due componenti distinti, e in pratica sono un'estensione dei concetti binders e functions attualmente in vigore nello Standard Library. Utilizzando la combinazione di queste due librerie abbiamo potuto trattare le funzioni passate ai Thread come oggetti, ciò ha semplificato notevolmente sia il loro

passaggio tra i vari membri delle classi che la leggibilità del codice.

3.2.2. Thread

La libreria in origine è stata sviluppata e progettata da William E. Kempf, e fa apparire l'utilizzo dei thread nella concorrenza in C ++ che in Java come una cosa semplice e intuitiva. E' questa la libreria su cui abbiamo basato il nostro sviluppo estendendola.

3.2.3. Espressioni regolari

Questa libreria ci mette a disposizione una serie di utility per la gestione delle espressioni regolari; la ricerca per mezzo delle espressioni regolari risulta molto utile nell'utility TraceString da noi sviluppata, permettendoci di individuare le occorrenze e seguire in modo agile le sequenze di esecuzione dei thread della fase di test.

3.3. Cosa ci fornisce ancora Boost

Facciamo una breve carrellata di alcuni altri componenti che fanno parte della libreria Boost e che sono tra i più diffusi nell'utilizzo dei programmatori. Per un'analisi più approfondita è possibile consultare la documentazione fornita dalle librerie stesse su internet (3).

3.3.1. Smart Pointers (Puntatori Intelligenti)

Sono strumenti che impediscono la perdita di risorse (in modo particolare se si hanno delle eccezioni), promuove il concetto di inizializzazione e acquisizione delle risorse.

Nelle librerie standard STD abbiamo lo strumento *std::auto_ptr* che però presenta diverse limitazioni:

- non può essere memorizzati all'interno di un contenitore standard
- non può essere facilmente utilizzato per implementare l'idioma pImpl (pointer-to-implementation)
- non funziona con gli array

I 5 tipi di puntatori Boost Intelligente superano questi difetti e forniscono molte funzioni extra tra cui:

- permettono la personalizzazione delle funzioni di cancellazione
- rilevano se un tipo template è incompleto

3.3.2. Composers

Functors e Binders sono diventati componenti comuni da usare nello STL, usando però le librerie più standard può risultare difficile combinare più funzioni. Composers permette ai funtori di essere combinati in diversi modi, riducendo al minimo la quantità di volte che gli utenti devono scrivere i loro loop.

3.3.3. Any

Questo componente fornisce un modo sicuro per spostare qualsiasi tipo di

componente, tutto ciò senza dover fare affidamento su puntatori nulli o unioni. Si tratta di un contenitore generico. Un oggetto any può contenere il valore di un qualsiasi tipo, senza effettuare alcun tipo di conversione. Una cosa simile alla boost::any la troviamo in Alexandrescu s Modern C++ Design nella guida sui Functors e sull'implementazione dei Functors.

3.3.4. Lambda Library

La libreria lambda fornisce una scorciatoia per la produzione di bind, functor e composers utilizzando le espressioni template.

3.3.5. The Boost Graph Library(Il BGL)

Il BGL è una libreria enorme, con una grande quantità di materiale di supporto e buoni programmi di esempi. La Biblioteca Graph Boost, Il Manuale dell'utente e Manuale di riferimento è stato pubblicato da Addison-Wesley The Boost Graph Library (La stessa fantastica serie che include 'Exceptional C + +', 'More Exceptional C + +' e 'Modern C++ Design'), che credo sia testimonianza della qualità della biblioteca.

3.3.6. Cosa Ancora

Traits (Tratti)

File System (Iterazione Directory)

Iterator Adaptors (Adattatore di Iteratore)

Maths and Matrices (Matematica e Matrici)

A Template metaprogramming framework

Tuples (Tuple)

Python

3.4. boost::thread

Analizziamo in modo più dettagliato la libreria che riguarda i thread.

```
#include <boost/thread.hpp>

boost::thread my_thread(my_function);
```

3.4.1. Chiamata di un Thread

Per creare un thread dobbiamo passare al costruttore della classe un oggetto *callable* cioè una funzione o un oggetto funzione. La funzione chiamabile viene copiata nell'oggetto Thread in modo tale da distruggerla in modo sicuro in seguito.

```
struct callable{
  void operator()(){…}
};
boost::thread ok_function() {
  callable x;
  return boost::thread(x);
} //x viene distrutto, ma era una copia, quindi è corretto

boost::thread bad_function() {
  callable x;
  return boost::thread(boost::ref(x));
}
// passo un riferimento a x, il riferimento viene copiato,
// ma poiché la x viene distrutta, il risultato è indefinite
```

Anche gli argomenti passati alla funzione che deve essere eseguita vengono copiati, ciò a meno che non siano dei riferimenti, questo è corretto in quanto in seguito voglio poterli distruggere:

```
class A { ... };

void myfun(A a) { /* thread body */ }

boost::thread f()
{
  A a;
  return boost::thread(myfunction, a);
}
```

Se dobbiamo passare un riferimento useremo la boost::ref().

Se una funzione o un oggetto callable passato alla boost::thread lancia un'eccezione che non è boost::thread_interrupted, il programma viene terminato:

```
void f () {
  throws "Questa è un\'eccezione";
}

void g () {
  boost::thread th(f);
```

```
  cout << " Questo non viene mai stampato " << endl;
  th.join();
}
```

3.4.2. Joining e Detaching

Nel seguente esempio vogliamo attendere la terminazione del thread

```
x = 0;
void long_thread() {
  for (long long i=0; i<100000000; i++);
  x = 1;
}

void make_thread()
{
  boost::thread th(long_thread);
}

int main() {
  make_thread();
  while (!x); // aspetto che il thread finisca
}
```

Se non attendiamo la fine del thread e l'oggetto che lo rappresenta viene distrutto, il thread si distacca. Un volta che un thread è distaccato, continuerà la propria esecuzione fino a quanto l'invocazione della funzione passata o l'oggetto chiamabile passato nella fase di costruzione del thread non terminano. Il thread può anche essere distaccato esplicitamente invocando la funzione *detach()* membro dell'oggetto *boost::thread*. In questo caso il thread cessa di rappresentare un thread distaccato e diventa un not-a-thread.

Nel precedente esempio attendevamo la terminazione di un thread per mezzo di una funzione while, l'attesa corretta deve avvenire per mezzo della funzione *join()* o *timed_join()* anch'esse membro della boost::thread. La funzione *join()* blocca il thread chiamante fino a quando il thread rappresentato dall'oggetto boost::thread specificato non viene completato. Se l'oggetto thread ha già terminato la propria esecuzione o l'oggetto indicato rappresenta un *not-a-thread* la funzione join() ritorna immediatamente. La funzione *timed_join()* è simile, l'unica variante è che la funzione join in questo caso ritorna dopo il tempo specificato anche se il thread non è terminato.

3.4.3. Interruzione

Un thread in esecuzione può essere interrotto invocando la funzione *interrupt()* sull'oggetto boost::thread. Quando su un thread viene invocata un'interruzione, questo (se ha le interruzioni abilitate) si interrompe in uno dei punti di interruzione specificati e genera un'eccezione nel thread interrotto.

Se l'eccezione non viene catturata, il thread termina, lo stack viene svuotato e vengono

invocati tutti i distruttori degli oggetti relativi.

Se vogliamo evitare di essere interrotti, possiamo creare un'istanza di un oggetto *boost::this_thread::disable_interruption*, quando tale oggetto viene distrutto, verrà ripristinato lo stato di protezione precedente:

```
void f()
{
   // interruzioni abilitate
   {
      boost::this_thread::disable_interruption di;
      // interruzioni disabilitate
      {
         boost::this_thread::disable_interruption di2;
         // interruzioni ancora disabilitate
      }    // di2 distrutto, ripristino stato interruzioni
      // interruzioni ancora disabilitate
   }   // di distrutto, ripristino stato interruzioni
   // interruzioni ora abilitate
}
```

Abbiamo a disposizione anche una funzione contraria alla disable_interruption, la *boost::this_thread::restore_interruption* che ha il seguente effetto:

```
void g()
{
   // Interruption Abilitate
   {
      boost::this_thread::disable_interruption di;
      // Interruption disabilitati
      {
         boost::this_thread::restore_interruption ri(di);
         // Interruption Abilitati
      }   // ri distrutto, Interruption ancora disabilitati
   }   // di distrutto, ripristino stato Interruption
   // Interruption ora Abilitati
}
```

Lo stato corrente delle interruzioni può essere richiamato con la seguente interrogazione *boost::this_thread::interruption_enabled()*.

3.4.4. Thread ID

L'oggetto *boost::thread::id* può essere utilizzato per identificare un thread. Ogni thread in esecuzione possiede un proprio ID univoco che può essere ricavato dall'oggetto thread per mezzo della funzione membro *get_id()* o chiamando la *boost::this_thread::get_id()* all'interno del thread. Gli oggetti boost::thread::id possono essere copiati e utilizzati all'interno di contenitori associativi in quanto viene fornita l'intera gamma degli operatori di confronto.

3.4.5. Class Thread

Con quanto detto la classe Thread è così definita:

```
#include <boost/thread/thread.hpp>

class thread
{
public:
    thread();
    ~thread();

    template <class F>
    explicit thread(F f);

    template <class F,class A1,class A2,...>
    thread(F f,A1 a1,A2 a2,...);

    template <class F>
    thread(detail::thread_move_t<F> f);

    // move support
    thread(detail::thread_move_t<thread> x);
    thread& operator=(detail::thread_move_t<thread> x);
    operator detail::thread_move_t<thread>();
    detail::thread_move_t<thread> move();

    void swap(thread& x);

    class id;
    id get_id() const;

    bool joinable() const;
    void join();
    bool timed_join(const system_time& wait_until);

    template<typename TimeDuration>
    bool timed_join(TimeDuration const& rel_time);

    void detach();

    static unsigned hardware_concurrency();

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    void interrupt();
    bool interruption_requested() const;

    // backwards compatibility
    bool operator==(const thread& other) const;
    bool operator!=(const thread& other) const;

    static void yield();
    static void sleep(const system_time& xt);
};
```

```
void swap(thread& lhs,thread& rhs);
detail::thread_move_t<thread>
move(detail::thread_move_t<thread> t);
```

3.4.6. Thread Group

E' possibile creare un gruppo di thread usando la classe *thread_group*, possiamo aggiungere un nuovo thread al gruppo usando la *add_thread()* e la funzione membro *create_thread()*. L'utilizzo dei gruppo è molto utile per attendere la terminazione di tutti i thread o interromperli tutti contemporaneamente tramite le rispettive funzioni *join_all()* e *interrupt_all()*.

La Classe rappresentativa è quindi la seguente:

```
#include <boost/thread/thread.hpp>

class thread_group:
    private noncopyable
{
public:
    thread_group();
    ~thread_group();

    template<typename F>
    thread* create_thread(F threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    void join_all();
    void interrupt_all();
    int size() const;
};
```

3.5. Mutex Concepts

Un oggetto mutex facilita la protezione contro le corse dai dati permettendo ai thread di gestire in modo concorrente i dati condivisi. Un Thread ottiene la funzionalità dell'oggetto mutex chiamando su di esso la relativa funzione di blocco *lock()* e la rilascia chiamando la funzione di sblocco *unlock()*. Un mutex può essere ricorsivo o non ricorsivo inoltre può concedere proprietà simultanee a uno o più thread.

Su un oggetto Boost:Thread possiamo invocare quattro tipi distinti di blocco che concettualmente possono essere classificati come:

- Lockable
- TimeLockable
- ShareLockable
- UpgradeLockable

3.5.1. Concetto Lockable

In questo caso l'oggetto acquisisce le proprietà di mutua esclusione vera e propria per

mezzo delle funzioni:

```
void lock();
bool try_lock();
void unlock();
```

La proprietà di blocco viene acquisita mediante la chiamata a *lock()* o *try_lock()* e rilasciata per mezzo della chiamata a *unlock()*.

3.5.2. Concetto TimedLockable

Il TimedLockable affina il concetto Lockable aggiungendo il supporto per i timeout quando si cerca di acquisire il blocco. Il tipo TimedLockable deve soddisfare i requisiti della Lockable, inoltre è possibile utilizzare la seguente funzione membro:

```
bool timed_lock(boost::system_time const& abs_time)
```

La funzione tenta di ottenere la proprietà di blocco per il thread corrente, il thread rimane bloccato finché il tempo specificato non viene raggiunto (il tempo specificato è di tipo assoluto), se il tempo specificato è già passato, la funzione si comporta come la *try_lock()*.

```
template<typename DurationType>
bool timed_lock(DurationType const& rel_time)
```

In questo caso la funzione rimane bloccata per un tempo relativo a partire dall'istante di chiamata, è come la funzione *timed_lock (boost::get_system_time() + rel_time)*.

In entrambi i casi per sbloccare la sincronizzazione prima del raggiungimento del tempo indicato, possiamo invocare la funzione *unlock()*.

3.5.3. Concetto SharedLockable

Il concetto SharedLockable è un perfezionamento del concetto TimedLockable, esso permette di impostare sia la proprietà di condivisione che la proprietà di mutua esclusione. Questo è lo standard utilizzato per il modello multi-reader/single-writer. Solo un thread alla volta può avere proprietà esclusiva e se ogni thread ha proprietà esclusive, allora nessun thread potrà essere condiviso. In alternativa tanti thread potranno avere contemporaneamente la proprietà di condivisione.

Oltre alle funzione membro della TimedLockable abbiamo a disposizione le seguenti funzioni membro:

```
void lock_shared()
```

Blocca il thread corrente finché la proprietà di condivisone può essere mantenuta sul thread stesso

```
bool try_lock_shared()
```

Tentativo di ottenere la proprietà condivisa per il thread corrente senza bloccare

```
bool timed_lock_shared(boost::system_time const& abs_time)
```

Tentativo di ottenere la proprietà condivisa per il thread corrente, il thread corrente rimane bloccato finché la proprietà di condivisione può essere rispettata oppure il tempo specificato viene raggiunto (il tempo indicato è da considerarsi assoluto). Se il tempo specificato è già passato, si comporta come la *try_lock_shared()*

```
void unlock_shared()
```

Le proprietà di blocco acquisite per mezzo delle funzioni precedenti devono essere rilasciate per mezzo della funzione *unlock_shared()*.

3.5.4. Concetto UpgradeLockable

Anche questo è un perfezionamento del precedente metodo, in questo caso sarà quindi possibile rispettare le proprietà di aggiornabilità, condivisione e esclusività. Questa è un'estensione per il modello multi-reader/single-writer fornito dal concetto SharedLockable: un singolo thread può avere proprietà di aggiornabilità e allo stesso tempo avere con altri thread delle proprietà comuni. Un thread con proprietà aggiornabile può tentare in qualsiasi momento un aggiornamento acquisendo la proprietà di esclusività. Se nessun altro thread ha richiesto la proprietà di condivisione, l'aggiornamento viene effettuato immediatamente, il thread acquisisce la proprietà di esclusività, per uscire da questa dovremo usare la *unlock()*, proprio come se la proprietà fosse stata acquisita con la *lock()*.

Se un thread con la proprietà aggiornabile tenta di aggiornare mentre altri thread hanno acquisito la proprietà di condivisione, il tentativo fallirà e il thread si bloccherà fino a quando la proprietà di esclusività non potrà essere acquisita.

Una proprietà può anche essere declassata e aggiornata:

- una proprietà esclusiva di un'implementazione con concetto UpgradeLockable può essere degradata alla proprietà aggiornabile
- una proprietà condivisa o aggiornabile può essere declassata a semplice proprietà di condivisione

In questo concetto oltre alle funzioni membro degli altri concetto troviamo le funzioni:

```
void lock_upgrade()
```

Blocco il thread finché non posso ottenere la proprietà di aggiornamento per il thread corrente. Quando mi sblocco ho ottenuto la proprietà di aggiornamento su *this

```
void unlock_upgrade()
```

In questo caso abbiamo già ottenuto la proprietà di aggiornamento su *this. La

funzione comunica che viene rilasciata la proprietà di aggiornamento sul thread corrente.

```
void unlock_upgrade_and_lock()
```

In questo caso si suppone di avere la proprietà di aggiornamento su *this, la funzione atomicamente comunica la proprietà di aggiornamento di *this dal thread corrente e acquisisce la proprietà esclusiva di *this. Se qualche altro thread ha la proprietà di condivisione, si blocca finché non può essere acquisita la proprietà di esclusività. Quando esco ho acquisito la proprietà esclusiva di *this

```
void unlock_upgrade_and_lock_shared()
```

In questo caso si suppone di avere la proprietà di aggiornamento di *this, la funzione comunica atomicamente la proprietà di aggiornamento di *this dal thread corrente e acquista la proprietà condivisa di *this senza bloccare. Quando esco il thread corrente ha ottenuto la proprietà di condivisione.

```
void unlock_and_lock_upgrade()
```

In questo caso si suppone di avere la proprietà esclusiva di *this, la funzione comunica automicamente la proprietà esclusiva di *this dal thread corrente e acquista la proprietà di aggiornamento *this senza bloccare. Quando esco il thread corrente ha la proprietà di aggiornamento *this.

La proprietà di blocco acquisito con la *lock_upgrade()* deve essere rilasciata dalla chiamata della funzione *unlock_upgrade()*. Se la proprietà del semaforo viene cambiata attraverso la chiamata a uno dei *unlock_xxx_and_lock_yyy()*, la proprietà deve essere rilasciata attraverso la chiamata alla funzione di sblocco corrispondente al nuovo livello di proprietà.

3.6. Variable Condition

Le classi *condition_variable* e *condition_variable_any* forniscono un meccanismo che permette ad un thread di attendere che un altro thread comunichi una notifica sull'avvenuta realizzazione di una condizione particolare. Il modello general prevede che un thread blocchi un mutex e quindi chiami la *wait()* su un'istanza di un oggetto *condition_variable* o *condition_variable_any*. Quando il thread è svegliato dall'attesa, verifica se la condizione considerata è vera, e se lo è continua l'esecuzione. Se la condizione non è vera, allora il thread chiama nuovamente la *wait()* per rimettersi in attesa. Nel caso più semplice la variabile è un semplice bool.

```
boost::condition_variable cond;
boost::mutex mut;
bool data_ready;

void process_data();
```

```
void wait_for_data_to_process()
{
    boost::unique_lock<boost::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}
```

Nel precedente esempio possiamo notare come il mutex *lock* viene passato alla *wait()* della variabile condizionale, la funzione wait infatti atomicamente aggiunge il thread all'insieme dei thread in attesa sulla variabile condizionale e sblocca il mutex. Quando il thread viene svegliato, il mutex tornerà ad essere bloccato nuovamente prima che la chiamata a *wait()* sia ritornata. Questo permette ad altri thread di acquisire il mutex per aggiornare i dati condivisi e garantire che i dati associati con la condizione siano correttamente sincronizzati.

Nel frattempo un altro thread imposterà la condizione di *true* e chiamerà o la *notify_one* o la *notify_all* sulla variabile condizionale in modo da svegliare rispettivamente un thread in attesa o tutti i thread in attesa.

```
void retrieve_data();
void prepare_data();

void prepare_data_for_processing()
{
    retrieve_data();
    prepare_data();
    {
        boost::lock_guard<boost::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}
```

Notiamo da questo secondo esempio come il mutex sia bloccato durante l'aggiornamento dei dati condivisi, ma che il mutex non deve essere bloccato durante la chiamata alla *notify_one()*.

Questo esempio utilizza un oggetto di tipo *condition_variable* ma avrebbe funzionato altrettanto bene con un oggetto di tipo *condition_variable_any*: questo è più generale, funziona con qualsiasi tipo di *lock* o *mutex*, mentre *condition_variable* richiede che sia già stato acquisito un *lock* prima dell'istruzione *wait()*. Ne è un esempio l'utilizzo della *boost::unique_lock < boost::mutex >*. Questa variante consente alla *condition_variable* di fare alcune ottimizzazioni in base alla conoscenza del tipo di mutex, in generale quindi l'utilizzo della *condition_varible_any* determina un'implementazione più

complessa rispetto all'utilizzo della *condition_variable*.

3.6.1. Funzioni Membro

```
void notify_one ()
```

Se alcuni thread sono attualmente bloccato in attesa di *this a causa di una chiamata alla *wait()* o alla *timed_wait()*, ne sblocca uno.

```
void notify_all ()
```

Se alcuni thread sono attualmente bloccati sul semaforo a causa di una chiamata alla *wait()* o alla *timed_wait()*, li sblocca tutti.

```
void wait ( boost :: unique_lock < boost :: mutex >& lock )
```

In questo caso supponiamo che il thread corrente sia già bloccato sul mutex lock e che nessun altro thread sia in attesa di questo. Atomicamente la funzione chiama la *lock.unlock()* e blocca il thread corrente. Il thread si sblocca nel caso di una notifica effettuata tramite la *this->notify_one()* o *this_notify_all()*. Quando il thread verrà sbloccato, il blocco viene riacquisito invocando la *lock.lock()* prima che la chiamata del *wait()* ritorni. Nel caso avvenga un'eccezione il *lock()* viene ugualmente ristabilito.

```
template < typename predicate_type >
void wait ( boost :: unique_lock < boost :: mutex >& lock ,
            predicate_type pred )
```

Equivale all'esecuzione di:

```
while (! pred ())
 {
     wait (lock);
 }

bool timed_wait ( boost :: unique_lock < boost :: mutex >& lock ,
                  boost :: system_time const & abs_time )
```

Si comporta come la *wait()*, il thread considerato però oltre alle precedenti condizioni di sblocco, si sbloccherà anche nel caso in cui il tempo attuale supera o sarò pari al valore del tempo assoluto indicato. La funzione quindi per indicare ciò ritornerà *false* se il tempo specificato da *abs_time* è stato superato, *true* altrimenti.

```
template < typename duration_type >
bool timed_wait ( boost :: unique_lock < boost :: mutex >& lock , duration_type const &
rel_time )
```

Si comporta come il precedente caso, qui però il tempo indicato è relativo invece che assoluto

```
template < typename predicate_type >
bool timed_wait ( boost :: unique_lock < boost :: mutex >& lock ,
```

```
boost :: system_time const & abs_time , predicate_type pred )
```

Questo caso equivale ad eseguire:

```
while(!pred())
{
   if(!timed_wait(lock,abs_time))
   {
    return pred();
   }
}
return true;
```

3.6.2. Classe Condition Variable

```
#include <boost/thread/condition_variable.hpp>

namespace boost
{
  class condition_variable
  {
  public:
    condition_variable();
    ~condition_variable();

    void notify_one();
    void notify_all();

    void wait(boost::unique_lock<boost::mutex>& lock);

    template<typename predicate_type>
    void wait(boost::unique_lock<boost::mutex>& lock,
              predicate_type predicate);

    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    boost::system_time const& abs_time);

    template<typename duration_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    duration_type const& rel_time);

    template<typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    boost::system_time const& abs_time,
                    predicate_type predicate);

    template<typename duration_type,
             typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    duration_type const& rel_time,
                    predicate_type predicate);

    // backwards compatibility

    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    boost::xtime const& abs_time);
```

```
    template<typename predicate_type>
    bool timed_wait(boost::unique_lock<boost::mutex>& lock,
                    boost::xtime const& abs_time,
                    predicate_type predicate);
  };
}
```

3.7. Barrier

Una barriera è un concetto semplice. Viene anche definito come un appuntamento, è un punto di sincronizzazione tra più thread. La barriera viene configurata per un determinato numero di thread(n), e quando ogni thread raggiunge la barriera deve attendere che tutti gli n thread siano arrivati. Una volta che l'n-esimo thread ha raggiunto la barriera, tutti i thread in attesa possono procedere e la barriera viene resettata.

```
# Include <boost / filo / barriera. HPP>

class barrier
{
public:
  barrier (unsigned int count);
  ~barrie();

  bool wait ();
};
```

Le istanze della barriera non sono copiabili o movibili.

La funzione *wait()* quindi blocca il thread corrente fino a quando tutti gli n thread non hanno invocato la *wait()*. Quando il thread n-esimo chiama la *wait()*, tutti i thread in attesa vengono sbloccati e la barriera viene resettata.

3.8. boost::regex

Le *Espressioni Regolari* sono una sequenza di simboli (quindi una stringa) che identifica un insieme di stringhe.

Una espressione regolare definisce una funzione che prende in ingresso una stringa e restituisce in uscita un valore di tipo si/no a seconda che la stringa segua o meno un certo pattern. Ad esempio, tutti gli indirizzi e-mail devono essere costituiti nel seguente modo: cominciare con una sequenza di caratteri alfanumerici, seguiti dal simbolo @, seguiti da altri caratteri alfanumerici, seguiti dal punto, seguiti da due lettere.

Le espressioni regolari sono composte da costanti e operatori che denotano insiemi di stringhe, e da operazioni tra gli insiemi.

Le espressioni regolari vengono utilizzate principalmente dagli editor di testo per la ricerca e la sostituzione di porzioni di testo.

E' possibile definire due tipologie di sintassi, base o estesa in entrambi i casi seguita da uno standard POSIX, è sufficiente specificare nella dichiarazione la proprietà:

```
// e1 è un caso di POSIX-Basic expression:
boost::regex e1(my_expression, boost::regex::basic);

// e2 è un caso di POSIX-Extended expression:
boost::regex e2(my_expression, boost::regex::extended);
```

Nella **sintassi estesa** delle regular expression tutti i caratteri corrispondono a se stessi, tranne i seguenti caratteri speciali:

```
.[{}() \ *+?|^$
```

3.8.1. Jolli "."

Il singolo carattere "." quando viene utilizzato al di fuori di un insieme di caratteri corrisponde ad ogni singolo carattere ad eccezione dei simboli:

- NULL quando il flag *match_no_dot_null* viene passato all'algoritmo di matching
- NUOVA RIGA quando il flag *match_not_dot_newline* viene passato all'algoritmo di matching

3.8.2. Ancora "^" , "$"

Un carattere "^" deve corrispondere all'inizio di una riga, quando viene utilizzato come primo carattere di un'espressione, oppure come primo carattere di una sotto-espressione.

Il carattere "$" invece deve corrispondere alla fine di una riga quando viene utilizzato come ultimo carattere di un'espressione, oppure come ultimo carattere di una sotto-espressione.

3.8.3. Contrassegno sub-expression (....)

Una sezione che inizia con "(" e finisce con ")" indica una sotto-espressione. Una sotto-espressione sarà quindi separata nella ricerca degli algoritmi di matching.

3.8.4. Ripetizione "*" , "+" , "?" , "{...}"

Ogni atomo (un singolo carattere, una sotto-espressione indicata o una classe di caratteri) possono essere ripetuti utilizzando gli operatori *, +, ?, { }.

* corrisponde all'atomo precedente zero o più volte, ad esempio a * b corrisponde a:

```
b    ab     aaaaaaaaaaab
```

+ corrisponde all'atomo precedente una o più volte, ad esempio a + b corrisponde a:

```
ab   aaaaaaaaaaaaab
```

? corrisponde all'atomo precedente zero o una volta, ad esempio ca ? b corrisponde a:

```
cb   cabina
```

{ } posso ripetere l'atomo in modo delimitato:

```
a { n }       corrisponde ad a ripetuto esattamente n volte
a { n ,}      corrisponde ad a ripetuto n o più volte
a { n , m }   corrisponde ad a ripetuto tra n ed m comprese
              volte
```

3.8.5. Alternanza "|"

L'operatore "|" corrisponde a uno dei suoi argomenti, ad esempio

```
abc | def    corrisponde a "abc" o a "def"
```

3.8.6. Set di Caratteri "[….]"

Un set di caratteri specifica un insieme di caratteri che inizia con "[" e terminano con "]", e corrisponde a qualsiasi singolo carattere che è membro di tale set. Lo possiamo quindi utilizzare nelle seguenti varianti:

- **Singolo Carattere**
  [abc] corrisponde a "a" o a "b" o a "c"
- **Range di Caratteri**
  [a-c] corrisponde ai singoli caratteri compresi tra "a" e "c"
- **Negazione**
  Se la prima espressione delle parentesi inizia con il carattere "^" allora corrisponde al complemento di ciò che è indicato, ad esempio l'espressione [^a-c] indica qualunque carattere diverso dall'intervallo tra a e c
- **Classe di Caratteri**
  Un'espressione della forma [[:name:]] corrisponde al nome di classe[7] "nome", ad esempio [[:lower:]] corrisponde a qualsiasi carattere minuscolo.
- **Elementi di Confronto**
  Un'espressione della forma [[.col.]] corrisponde alla collezione di elementi di nome *"col"*.
- Elementi di Equivalenza
  Un'espressione della forma [[=col=]] corrisponde a qualsiasi carattere o elemento di confronto la cui prima chiave di ordinamento è la stessa di quella per la raccolta di elementi *"col"*.

3.8.7. Precedenza tra Operatori

L'ordine di precedenza tra gli operatori è la seguente:

1. Collezione relativa ai simboli parentesi [==] [::] [..]
2. Caratteri di escape \

[7] Per vedere le tipologie di classi di caratteri controllare l'appendice

3. Set di caratteri (espressione tra parentesi quadre) []
4. Raggruppamento ()
5. Singolo-carattere-ERE duplicazione * + ? { m , n }
6. Concatenazione
7. Ancoraggio ^ $
8. Alternanza |

3.8.8. Find String

L'interesse nel nostro caso sarà quello di ricercare una determinata espressione regolare in una stringa.

Per vedere come questa libreria può essere utilizzata supponiamo che stiamo scrivendo i numeri di una carta di credito e che ne vogliamo controllare la correttezza della digitazione. In genere il numero di una carta di credito è composta da 16 cifre suddiviso in gruppi di 4 separate da uno spazio o da un trattino. Una possibile espressione regolare che li rappresenta è la seguente:

```
(\\d{4}[- ]){3}\\d{4}
```

Per avere la convalida sarà sufficiente eseguire la seguente funzione:

```
bool validate_card_format(const std:string& s)
{
  static const boost::regex e("(\\d{4}[- ]){3}\\d{4}");
  return regex_match(s, e);
}
```

La funzione *regex_match()* si occuperà di verificare che all'interno della stringa s sia rispettata la regular expression e.

3.9. boost::function

La boost::function contiene una famiglia di classi template che ci permettono di creare un oggetto funzione. Il concetto è simile a una callback generalizzata. In generale possiamo utilizzare la boost::function in qualunque luogo dove avremmo utilizzato un puntatore a funzione. Ciò permette all'utente una maggiore flessibilità nella realizzazione del suo obiettivo.

La boost::function ha due forme sintattiche: la forma preferita e la forma compatibile. La forma preferita è più adatta al linguaggio C++ e riduce il numero dei parametri template che devono essere utilizzati, migliorandone spesso la leggibilità, questa forma però non è supportata su tutte le piattaforme. La forma compatibile funziona su tutti i compilatori supportati dalla boost::function.

La sintassi preferred sarà del tipo:

```
boost::function<float (int x, int y)> my_function;
```

dove abbiamo indicato che la funzione *my_function* ritornerà una variabile *float* e accetterà come ingresso due valori *interi*.

Nel caso di funzioni che ritornano void e non accettano parametri di ingresso avremo:

```
boost::function<void (void)> my_function;
```

3.10. boost::bind

La boost::bind è una generalizzazione delle funzioni standard std::bind1st e std::bind2nd. Supporta arbitrarie funzioni oggetto, puntatori a funzione, puntatori a funzioni membro; riesce inoltre ad associare qualsiasi argomento, sia questo un valore o un argomento, indipendentemente dalla posizione in cui si trova. La bind non pone requisiti sull'oggetto funzione, in particolare non richiede il tipo del risultato, il tipo del primo-secondo argomento ecc…

3.10.1. Utilizzo di bind con funzioni e puntatori a funzione

Definendo le seguenti funzioni:

```
int f(int a, int b)
{
    return a + b;
}

int g(int a, int b, int c)
{
    return a + b + c;
}
```

possiamo affermare che la chiamata a *bind(f, 1, 2)* produrrà un oggetto a funzione *"nullary"* che non accetta argomenti di ingresso e restituisce la chiamata *f(1, 2)*. Allo stesso modo *bind(g, 1, 2, 3)* sarà equivalente a g(1, 2, 3).

Possiamo anche legare selettivamente alcuni argomenti ad esempio bind(f, _1, 5)(x) è equivalente a f(x, 5). L'argomento _1 rappresenta a tutti gli effetti un segnaposto che indica "sostituire con il primo argomento che viene passato in ingresso".

Possiamo quindi anche scrivere:

```
bind(f, _2, _1)(x, y);                  // f(y, x)
bind(g, _1, 9, _1)(x);                  // g(x, 9, x)
bind(g, _3, _3, _3)(x, y, z);           // g(z, z, z)
bind(g, _1, _1, _1)(x, y, z);           // g(x, x, x)
```

3.10.2. Utilizzo di bind con oggetti funzione

Bind non è limitato all'utilizzo nelle funzioni, esso accetta anche arbitrari oggetti funzione. Nel caso generale il tipo di ritorno generato dalla funzione oggetto deve essere specificato. Vediamo un esempio:

```
struct F
{
    int operator()(int a, int b) { return a - b; }
    bool operator()(long a, long b) { return a == b; }
};

F f;

int x = 104;

bind<int>(f, _1, _1)(x);           // f(x, x), i.e. zero
```

3.10.3. Utilizzo di bind con puntatori a funzioni membro

Puntatori a funzioni membro e puntatori a dati membro non sono oggetti funzione, in quanto non supportano l'operatore (). Per comodità, bind accetta puntatori a funzioni membro come primo argomento: sarà come se avessimo utilizzato la *boost::mem_fn* per convertire il puntatore del membro in un oggetto funzione.

In altre parole vale la seguente equivalenza:

```
bind(&X::f, args)              bind<R>(mem_fn(&X::f), args)
```

Dove con R abbiamo indicato il tipo ritornato dalla funzione membro X::f .

3.10.4. Utilizzo di bind con function

Un comodo utilizzo di bind legato a function è indicato di seguito:

```
class button
{
public:

    boost::function<void()> onClick;
};

class player
{
public:

    void play();
    void stop();
};

button playButton, stopButton;
player thePlayer;

void connect()
{
    playButton.onClick =
                boost::bind(&player::play, &thePlayer);
    stopButton.onClick =
                boost::bind(&player::stop, &thePlayer);
}
```

Cap. 4 - GoogleTest

GoogleTest offre la possibilità di scrivere Test C++ su una varietà di piattaforme (Linux, Mac OS X, Windows, Cygwin, Windows CE, Symbian). E' basato su architettura xUnit.

Vediamo ciò che fa di GoogleTest un ottimo aiuto per le fasi di testing:

- I test dovrebbero essere **indipendenti e ripetibili.** GoogleTest isola il test eseguendo ciascuno di essi su un oggetto diverso. Quando un test fallisce, GoogleTest ci consente di eseguire un isolamento del debug veloce.
- I test dovrebbero essere **ben organizzati e riflettere la struttura del codice testato.** GoogleTest raggruppa i test in casi di test che sono in grado di condividere i dati e le subroutine.
- **I test dovrebbero essere portatiti e riutilizzabili.** I test devono essere indipendenti dalla piattaforma che li ospita. GoogleTest funziona su diversi sistemi operativi con diversi compilatori e con o senza l'utilizzo delle eccezioni.
- Quando i test non riescono **dovrebbero fornire il più possibile informazioni sul problema riscontrato.** GoogleTest non si ferma per la mancata correttezza di un test ma continua oltre.
- **Ogni test dovrebbe enumerarsi e indicare la correttezza del proprio risultato**. GoogleTest tiene automaticamente traccia di tutti i test definiti.
- **I test dovrebbero essere veloci.** GoogleTest prevede la possibilità di riutilizzare le risorse condivise attraverso i test e pagare per il set-up/tear-down una sola volta.

4.1. Concetti base

Quando si utilizza GoogleTest, scriviamo delle Asserzioni, queste sono delle affermazioni che verificano se una condizione è vera. Il risultato di un'affermazione può essere *successo, nonfatal failure* o *fatal failure*. Se avviene un errore irreversibile, si interrompe la funzione corrente, altrimenti la funzione continua normalmente.

Tests usa le assertions per verificare il comportamento del codice testato. Se il test va in crash o determina un'asserzione non riuscita, genera un *fallimento* altrimenti un *successo*.

Un Test case contiene uno o più test. Si consiglia di raggruppare i test in casi di test che riflettono la struttura stessa del codice testato. Questi possono essere contenuti in un programma apposito di test.

4.2. Asserzioni

Le affermazioni in GoogleTest sono delle macro che assomigliano a delle chiamate di

funzione. Si testa una classe o una funzione facendo delle affermazioni circa il suo comportamento. Quando un'asserzione fallisce, GoogleTest stampa il nome del file e il numero della linea dove si trova l'affermazione stessa insieme ad un messaggio di errore (tale messaggio potrà anche essere personalizzato).

Le affermazioni possono essere scritte in due modi apparentemente identici, per mezzo della ASSERT_* che quando fallisce interrompe la funzione corrente, o EXPECT_* che non interrompe la funzione corrente. E' quindi preferibile la forma EXPECT_* in quanto possiamo notare più errori contemporanei nel medesimo test, è comunque preferibile la ASSERT_* nel caso in cui non abbia senso continuare il flusso di controllo.

Per fornire un messaggio personalizzato in caso di fallimento del test è sufficiente utilizzare l'operatore << a seguito della macro. Vediamo un esempio:

```
ASSERT_EQ(x.size(), y.size())
  << "Il vettore x e y non hanno la stessa lunghezza ";

for (int i = 0; i < x.size(); ++i) {
  EXPECT_EQ(x[i], y[i])
    << "Il vettore x e y differiscono per indice" << i;
}
```

Vediamo una serie di affermazioni che sono disponibili:

AFFERMAZIONE FATALE	AFFERMAZIONE NON FATALE	VERIFICA
ASSERT_TRUE (condizione);	EXPECT_TRUE (condizione);	condizione è vera
ASSERT_FALSE (condizione);	EXPECT_FALSE (condizione);	condizione è falsa

AFFERMAZIONE FATALE	AFFERMAZIONE NON FATALE	VERIFICA
ASSERT_EQ (previsto, effettivo);	EXPECT_EQ (previsto, effettivo);	previsto == reale
ASSERT_NE (val1, val2);	EXPECT_NE (val1, val2);	val1! = val2
ASSERT_LT (val1, val2);	EXPECT_LT (val1, val2);	val1 <val2
ASSERT_LE (val1, val2);	EXPECT_LE (val1, val2);	val1 <= val2
ASSERT_GT (val1, val2);	EXPECT_GT (val1, val2);	val1> val2
ASSERT_GE (val1, val2);	EXPECT_GE (val1, val2);	val1> = val2

AFFERMAZIONE FATALE	AFFERMAZIONE NON FATALE	VERIFICA
ASSERT_STREQ (expected_str, actual_str);	EXPECT_STREQ (expected_str, actual_str);	le due stringhe C hanno lo stesso contenuto
ASSERT_STRNE (str1, str2);	EXPECT_STRNE (str1, str2);	le due stringhe C hanno contenuti diversi
ASSERT_STRCASEEQ (expected_str, actual_str);	EXPECT_STRCASEEQ (expected_str, actual_str);	le due stringhe C hanno lo stesso contenuto, caso ignorato
ASSERT_STRCASENE (str1, str2);	EXPECT_STRCASENE (str1, str2);	le due stringhe C hanno contenuto diverso, caso ignorato

4.3. Creare un semplice Test

Per creare un test si usa la macro TEST() dove dobbiamo definire e nominare la funzione di test. Questa è un'ordinaria funzione C++ che non ritorna niente.

```
TEST (test_case_name, nome_test) {
  ...  prova del corpo ...
 }
```

Il valore di *test_case_name* indica il nome del banco di prova e identifica il gruppo di appartenenza del test: se si utilizza lo stesso nome per più test questi verranno considerati come tutti appartenenti allo stesso gruppo.

Il valore *nome_test* indica il nome specifico del test e per ogni gruppo dovrà essere univoco.

In questa funzione possiamo includere, oltre a qualunque dichiarazione C++ valida, le affermazioni di GoogleTest in modo da verificare la correttezza del test svolto.

Il risultato del TEST è determinato dal risultato delle asserzioni che vi sono all'interno: se tutte hanno successo, allora il test ha successo se una sola fallisce il test fallisce.

4.4. Test Fixtures: Utilizzare la stesso configurazione di dati per prove multiple

Se dobbiamo scrivere due o più test che operano su dati simili, è possibile utilizzare un *dispositivo di prova*. Questo consente di riutilizzare la stessa configurazione di oggetti per diversi test.

Per creare un dispositivo di prova è sufficiente:

- derivare una classe da *::testing::Test* e inizializzarne il corpo in modo *protected* o *public* in base a come vogliamo accedervi;
- dichiarare all'interno della classe tutti gli oggetti che si intende utilizzare;
- scrivere, se necessario, un costruttore di default o la funzione *SetUp()* per preparare gli oggetti per ogni test.;
- scrivere, se necessario, una funzione distruttore o *TearDown()* per liberare tutte le risorse assegnate dal programma di SetUp();
- scrivere, se necessario, delle funzioni membro comuni per i test.

Se utilizziamo la TEST_F() invece della TEST() potremo accedere tranquillamente agli oggetti e alle subroutines del dispositivo di prova.

Per ogni TEST_F definito, GoogleTest esegue la seguente sequenza di operazioni:

1. crea un nuovo dispositivo di prova per il test;
2. lo inizializza immediatamente per mezzo della SetUp();

3. esegue il Test;
4. pulisce l'oggetto di prova chiamando la TearDown();
5. cancella il dispositivo di prova.

4.5. Invocare il Test

Entrambe le macro TEST() e TEST_F() registrano implicitamente le loro prove in GoogleTest. Non c'è quindi bisogno di andare ad aggiungere il test scritto da nessuna parte.

Una volta definito un test questo può essere mandato in esecuzione per mezzo della chiamata alla macro RUN_ALL_TEST() che restituisce 0 se tutti i test hanno successo, 1 altrimenti. La chiamata a questa macro gestisce tutti i test presenti e collegati alla nostra unità.

La chiamata alla macro RUN_ALL_TESTS() determina il seguente flusso di esecuzione:

1. salva lo stato di tutti i flags di GoogleTest;
2. crea un oggetto dispositivo di prova per il primo test;
3. lo inizializza per mezzo dell'eventuale SetUp();
4. esegue il test sul dispositivo di prova;
5. pulisce il dispositivo di prova chiamando la TearDown();
6. cancella il dispositivo di prova;
7. ripristina lo stato di tutti i flag di GoogleTest;
8. ripete tutti i precedenti passaggi per i test successivi fino a quanto non sono stati tutti eseguiti.

4.6. Scriviamo un esempio di funzione Main

```
#include "this/package/foo.h"
#include "gtest/gtest.h"

namespace {

  // Dispositivo di prova per il test della classe foo.
  class FooTest : public ::testing::Test {
  protected:
    // E' possibile rimuovere una o tutte le seguenti funzioni se il suo corpo è vuoto.

    FooTest() {
      // Si può fare il SetUp() per ogni test qui.
    }
    virtual ~FooTest() {
      // Si può distruggere ciò che non genera eccezioni qui.
    }

    // Se il costruttore e il distruttore non sono sufficienti per la creazione
    // e per la pulizia  di ogni test, è possibile definire i seguenti metodi
```

```
    virtual void SetUp() {
      // Questo codice sarà chiamato subito dopo il costruttore e prima di ogni test.
    }

    virtual void TearDown() {
      // Questo codice sarà chiamato prima del distruttore ma alla fine di ogni test.
    }

  // Gli oggetti qui dichiarati possono essere utilizzati da tutti i test del banco di prova
  // di della classe foo.
  };

  // Test per il metodo Foo::Bar() risulta Abc.
  TEST_F(FooTest, MethodBarDoesAbc) {
    const string input_filepath =
                 "this/package/testdata/myinputfile.dat";
    const string output_filepath =
                 "this/package/testdata/myoutputfile.dat";
    Foo f;
    EXPECT_EQ(0, f.Bar(input_filepath, output_filepath));
  }

  // Test per foo risulta Xyz.
  TEST_F(FooTest, DoesXyz) {
    // Exercises the Xyz feature of Foo.
  }

}  // namespace

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Ulteriori approfondimenti sono disponibili sul sito stesso della libreria GoogleTest (4).

Cap. 5 - Progettazione libreria

L'idea di sviluppo della nostra libreria ThreadUtility nasce avendo considerato la potenzialità e gli apprezzamenti nei riguardi delle librerie boost. Ciò può essere riscontrato facilmente in quanto alcune di esse andranno a far parte del nuovo standard C++ prossimamente rilasciato. Nelle librerie boost abbiamo però notato la mancanza di uno sviluppo nei confronti dei sistemi RealTime; da qui la nostra idea:

"cerchiamo di creare una libreria con delle utility che ci permettano di rendere più facile la gestione e l'implementazione dei sistemi RealTime".

Siamo quindi riusciti a realizzare:

- una classe Thread dove possiamo impostare i parametri di schedulazione indipendentemente dalla piattaforma (windows, Linux) utilizzata e sincronizzarne lo start;
- una classe che implementa un Thread Periodico RealTime modulare dove ogni modulo è virtuale e può quindi essere ridefinito ampliandone la riusabilità, il Thread può essere inoltre sincronizzato sia in fase di start che di end, ciò ci permette di realizzare delle strutture più complesse come le catene di thread;
- una struttura che gestisce i Cambi di Modo di lavoro dei Task
- altre utility che ci hanno permesso di facilitare la fase di test della nostra libreria e che possono risultare utili in altre diverse occasioni.

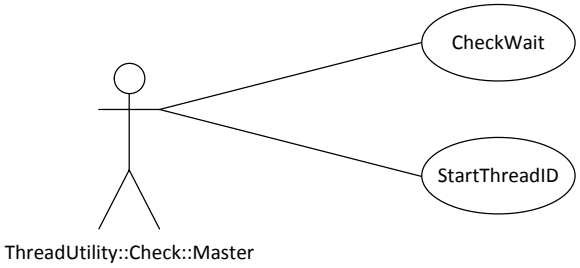
Iniziamo la nostra analisi di progetto proprio da queste utility (**Check**, **TraceString**, **Syncro**, **ClockTime**) e successivamente procederemo nell'analisi della struttura dei **Thread** che si è evoluta con metodo Botton-Up fino allo sviluppo dei **PeriodicRTThread** e dei **Cambi di Modo**.

5.1. Check

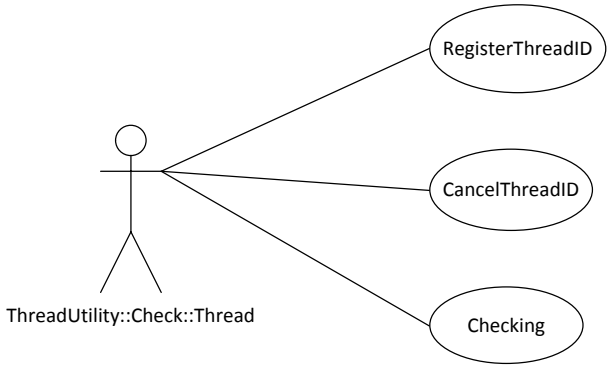
L'idea di questa utility nasce dalla necessità, in modo particolare nella fase di Test, di sincronizzare più flussi di codice che si trovano in esecuzione su thread differenti. Se vengono infatti eseguite diverse funzioni su thread differenti come ben sappiamo queste andranno in esecuzione in modo concorrente, e quindi non potremo chiarire il concetto di "chi arriva prima dove". La classe Check ci permette di settare dei punti di sincronizzazione su cui il flusso si interromperà in attesa che il master lo riattivi.

5.1.1. Casi d'Uso

Possiamo quindi individuare due tipologie di utenze che accedono a questa struttura, l'utente **Master** e l'utente **Thread**.



L'utente Master è quello che si pone in attesa dell'arrivo di un numero specificato *n* di CHECK da parte di utenti Thread (*CheckWait*), il Master quindi si sbloccherà dallo stato di attesa solo quando *n* utenti Thread saranno arrivati al punto di sincronizzazione indicato dalla Checking. Dopo lo sblocco il Master potrà attivare il thread da lui desiderato e questo ci permette, dopo l'avvenuta sincronizzazione, di decidere il successivo ordine di startup dei thread che erano giunti al punto di sincronizzazione (*StartThreadID*).



L'utente Thread sarà quello che sceglie dove porre i punti di sincronizzazione (*Checking*), quello cioè che decide dove il sistema si ferma in attesa che il master lo riattivi. Chiaramente per svolgere tutte questa funzione il Thread dovrà prima registrarsi alla struttura gestita dal Master (*RegisterThreadID*) e quindi alla fine dell'utilizzo cancellarsi (*CancelThreadID*).

5.1.2. Struttura

La classe Check è stata quindi strutturata utilizzando un oggetto mappa dove associamo l'identificatore del thread a una variabile condizionale (*MAPPA*), quando l'utente Thread si registra alla struttura Check non farà altro che aggiungere un nuovo elemento alla mappa e quando il Thread si vorrà cancellare non farà altro che eliminarlo.

```
typedef boost::thread::id Thread_ID;
typedef std::map<Thread_ID, boost::condition_variable*> MAPPA;
```

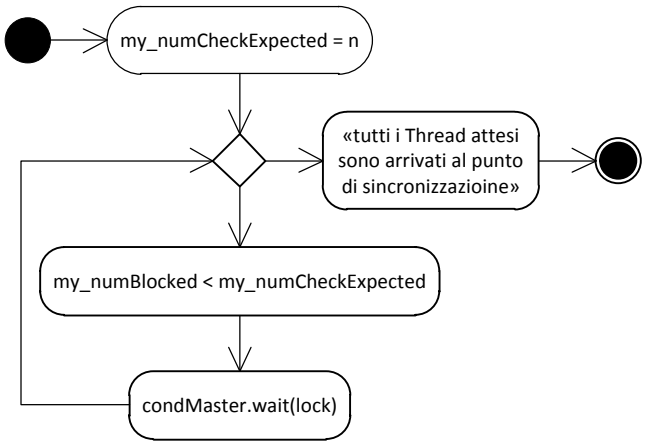
Check
-exclusive : mutex -my_signal : SyncroSignal -my_recorded : MAPPA -my_numRegistered : int -my_numBlocked : int -my_numCheckExpected : int
+Check() +~Check() +RegisterThread(in thID : Thread_ID) : bool +FindRegistration(in thID : Thread_ID) : bool +CancelThread(in thID : Thread_ID) : bool +Checking() : bool +CheckWait(in n : int) : int +CheckWait() : int +Start(in thID : Thread_ID) : bool +GetBlocked() : int +GetRegistered() : int

Dove le variabili:

- *exclusive:* viene utilizzato per l'uso esclusivo della risorsa
- *my_signal:* sincronizza il Master con l'arrivo di un nuovo Check
- *my_recorded:* è la mappa che associa i ThreadID dei Thread registrati alla relativa variabile condizionale che utilizziamo per sincronizzarci
- *my_numRegistered:* indica il numero di Thread che si sono registrati
- *my_numBlocked:* indica il numero di Thread che sono attualmente bloccati sulla sincronizzazioine
- *my_numCheckExpected:* sono i Thread che ho richiesto di attendere

Analizziamo ora le Attività indicate nei Casi d'Uso:

5.1.3. Attività – Checkwait(n)



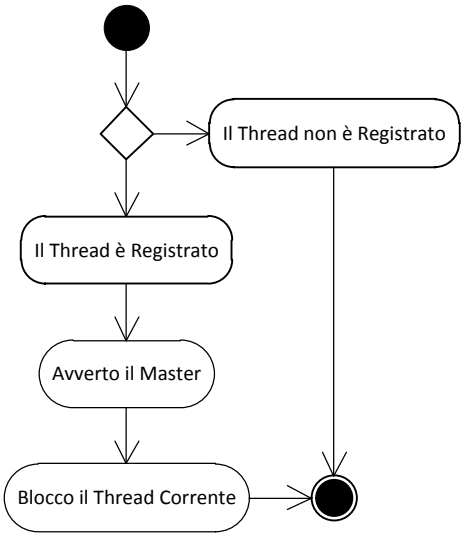
La funzione imposta il numero di elementi attesi con il valore *n* che è stato passato alla funzione. Se tutti i Thread sono arrivati al punto di sincronizzazione allora termino (mi sblocco ed esco dalla funzione), altrimenti se non abbiamo ancora raggiunto *n* punti di sincronizzazione ci blocchiamo in attesa che un altro Thread giunga al punto di sincronizzazione e ci risvegli.

5.1.4. Attività – Start(Thread_ID)

Se l'id passato alla funzione è registrato nella mappa allora lo sblocco.

```
if(!FindRegistration(thID)) return false;

my_numBlocked--;
my_recorded[thID]->notify_one();
```

5.1.5. Attività – Checking()



Se il Thread è registrato nella mappa avverto il Master indicandogli che sono arrivato al punto di sincronizzazione, questo si sveglierà ed eseguirà quanto indicato nella *CheckWait(n)*. Dopo aver avvertito il Master, il Thread si addormenta, sarà il master con la funzione *Start(ThreadID)* a risvegliare successivamente il Thread.

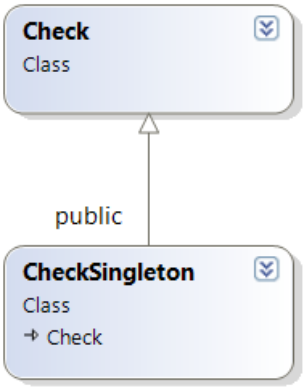
5.1.6. Evoluzione

Lo sviluppo è andato oltre e sono state aggiunte le seguenti proprietà:

- *Singleton:* la struttura Check deve essere unica nella gestione di un processo, non dobbiamo quindi dichiarare un oggetto Check e lavorare su di esso ma semplicemente invocare l'istanza dell'oggetto Singleton e su di esso invocare le funzioni membro desiderate (5).

```
CheckSingleton::Instance()
```

L'oggetto Singleton lo abbiamo ottenuto derivandolo dalla Check.



- *Meccanismo automatico per la registrazione e cancellazione su CheckSingleton*: quando dichiaro un oggetto CheckObject questo registra automaticamente il Thread su cui viene eseguito nella struttura Singleton della classe Check, quando l'oggetto viene distrutto, il Thread corrente viene anche automaticamente cancellato
- *Semplificazione di Utilizzo:* sono state definite delle macro che ci semplificano la scrittura:

```
#define CHK CheckSingleton::Instance()
#define CHECK CheckSingleton::Instance().Checking()
```

- *Utilizzo solo nella fase di Test*: la struttura Check sarà principalmente utilizzato nella fase di test dei Thread, vogliamo quindi poter eliminare a nostro piacimento i punti di sincronizzazione in modo che il flusso del Thread non venga più interrotto. Questo è realizzabile per mezzo di una semplice macro:

```
#ifdef CHECKTEST
    #define CHECK CheckSingleton::Instance().Checking()
#else
    #define CHECK
#endif
```

  Se nel master definiamo CHECKTEST allora i punti di sincronizzazione saranno attivi, se non lo definiamo non verrà eseguito niente e quindi la funzione non si bloccherà.

5.1.7. Esempio di Utilizzo

Quella che segue è la funzione che passeremo al Thread, come possiamo vedere abbiamo semplicemente dichiarato un oggetto CheckObject e successivamente abbiamo inserito due punti di sincronizzazione.

```
#include "../CheckSingleton.h"

void
BodyTest1()
{
  CheckObject sinc;
  cout << "Th 1: Start" << endl;

  cout << "Th 1: Prima Sincronizzazione" << endl;
  CHECK;

  cout << "Th 1: Seconda Sincronizzazione" << endl;
  CHECK;

  cout << "Th 1: Fine" << endl;
}
```

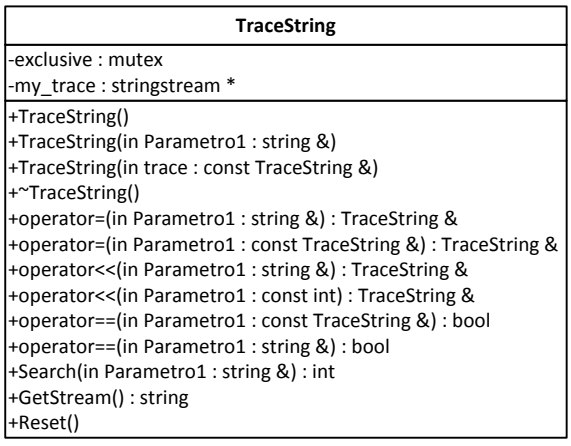
La funzione seguente è quella utilizzata nella fase di Test, come possiamo notare abbiamo semplicemente creato il Thread e successivamente ci siamo messi in attesa

del primo punto di sincronizzazione, abbiamo startato la funzione e poi ci siamo messi nuovamente in attesa di tutti i thread registrati, in questo caso è presente un solo thread, abbiamo ristartato il thread e quindi terminiamo.

```
TEST_F(CheckSingletonTest, UnThread)
{
  cout << "thread MAIN" << endl;
  ASSERT_EQ(CHK.GetRegistered(), 0);

  boost::thread th1(BodyTest1);
  CHK.CheckWait(1);

  cout << "thread MAIN" << endl;
  ASSERT_EQ(CHK.GetBlocked(), 1);
  ASSERT_EQ(CHK.GetRegistered(), 1);

  CHK.Start(th1.get_id());        // primo step

  CHK.CheckWait();

  ASSERT_EQ(CHK.GetBlocked(), 1);
  cout << "thread MAIN" << endl;

  CHK.Start(th1.get_id());        // secondo step
  th1.join();
  cout << "Master: Sono rientrato dal Thread 1" << endl;
  ASSERT_EQ(CHK.GetBlocked(), 0);
  ASSERT_EQ(CHK.GetRegistered(), 0);
}
```

La fase di test ci mostra che la struttura Check alla fine è vuota, ciò dimostra che anche l'oggetto CheckObject funziona correttamente.

5.2. TraceString



La Classe ci permette di scrivere in mutua esclusione su una stringa traccia, siano questi valori *int* o valori *stringa*. Si deve instanziare l'oggetto TraceString che può anche essere subito inizializzato con una stringa:

```
TraceString trace;
TraceString trace("stringa");
```

L'inserimento degli elementi nella traccia avviene per mezzo dell'operatore << :

```
trace << "Stringa";
trace << 1;
```

Come istruzioni di controllo possiamo utilizzare l'operatore ==

```
trace == "Stringa";
trace == trace;
```

Possiamo ritornare la stringa memorizzata per mezzo dell'istruzione GetStream()

```
trace.GetStream();
```

e resettare la stringa per mezzo di Reset()

```
trace.Reset();
```

E' di particolare interesse la funzione membro Search(const std:string&) che ritorna il numero di occorrenze della stringa che gli è stata passata ritrovate nella traccia, la stringa che noi passiamo viene trattata utilizzando le regolar expression.

```
Search(const std:string&)
```

5.2.1. Test

Scrivo nella traccia 10 X e 5 Y, successivamente controllo la correttezza della stringa per mezzo dell'operatore ==. Successivamente per mezzo della Search() conto il numero di X e di Y presenti nella traccia.

```
TEST_F(TraceStringTest, Scrivo_10X_e_controllo_che_ci_Siano)
{
  cout << "Scrivo nella Traccia le 10 X e 5 Y" << endl;
  cout << "Controllo che ci siano le 10 X ";
  cout << "e successivamente le 5 Y" << endl;
  for (int i=0; i<5; i++)
  {
    my_trace << "X";
    my_trace << "Y";
  }
  for (int i=0; i<5; i++)
    my_trace << "X";

  cout << "Ricerca: " << my_trace.Search("X") << endl;

  ASSERT_TRUE(my_trace.GetStream() == "XYXYXYXYXYXXXXX");
  cout << "Nella Traccia c'è una X" << endl;
  ASSERT_TRUE(my_trace.Search("X"));
  cout << "Nella Traccia ci sono 10 X" << endl;
  ASSERT_TRUE(my_trace.Search("X") == 10);
  cout << "Nella Traccia ci sono 5 Y" << endl;
  ASSERT_TRUE(my_trace.Search("Y") == 5);
}
```

5.3. ClockTime

Con la classe ClockTime abbiamo ridefinito le operazioni necessarie per la trattazione del tempo in modo assoluto. Abbiamo ridefinito le operazioni di confronto, di somma e di assegnamento.

ClockTime
-time : xtime
+ClockTime() +ClockTime(in us : long) +ClockTime(in tmp : const ClockTime &) +operator=(in Parametro1 : const ClockTime &) : ClockTime & +Get_SystemTime() : xtime +Get() : xtime +Get_MicroSec() : long +operator+=(in us : long) : ClockTime & +operator-=(in b : ClockTime &) : ClockTime & +operator<(in b : ClockTime &) : bool +operator<=(in b : ClockTime &) : bool +operator==(in b : ClockTime &) : bool +operator>(in b : ClockTime &) : bool +operator>=(in b : ClockTime &) : bool

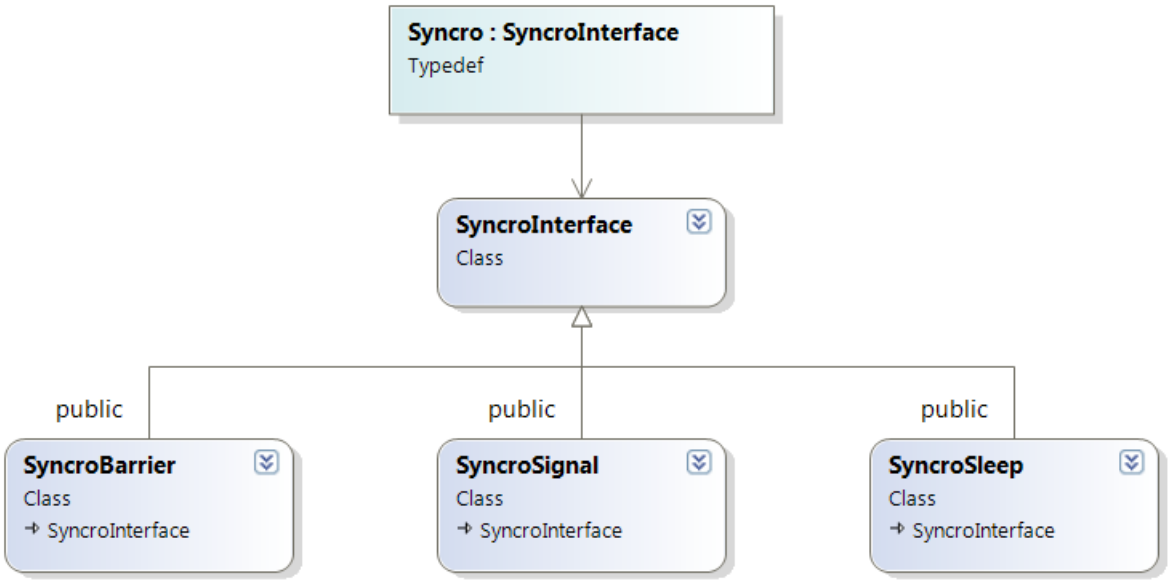
La maggioranza delle funzioni membro sono banali e non hanno bisogno di spiegazioni. Le funzioni più interessanti risultano:

- **Get_SystemTime()** che ritorna all'oggetto corrente il tempo assoluto attuale;
- **Get()** che ritorna l'oggetto boost::xtime rappresentato;
- **Get_MicroSec()** che ritorna il tempo rappresentato in microsecondi.

5.4. Syncro

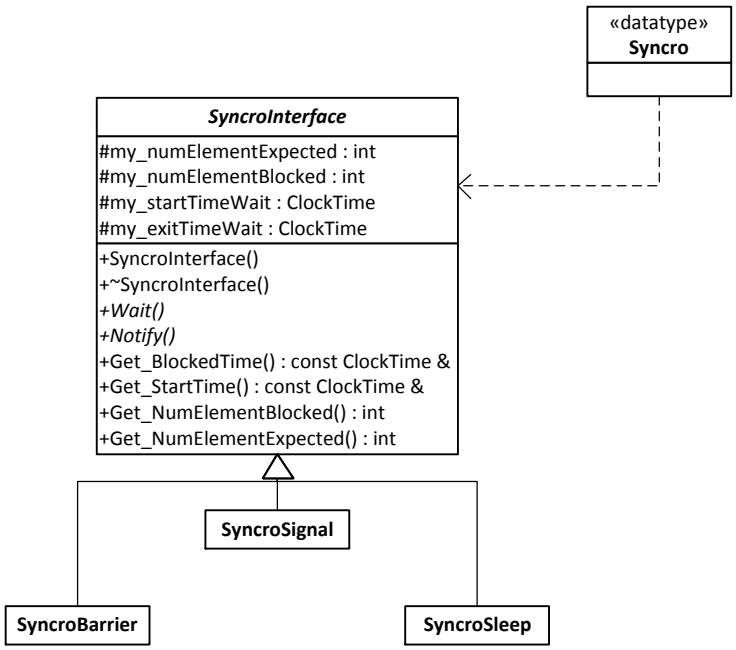
Syncro definisce un'interfaccia standard per l'implementazione dei semafori.

Dall'interfaccia deriveremo tutte le istanze dei semafori SyncroSleep, SyncroBarrier, SyncroSignal.



5.4.1. SyncroInterface

L'interfaccia contiene al suo interno le variabili comuni ai semafori, vengono però dichiarate virtuale pure le funzioni *Wait()* e *Notify()* che dovranno obbligatoriamente essere implementate nelle classi derivate in quanto specifiche della tipologia stessa del semaforo.



L'oggetto my_*startTimeWait* indica l'istante in cui viene chiamata la funzione Wait(), l'oggetto my_*exitTimeWait* indica l'istante in cui si esce dalla funzione Wait().

5.4.2. SyncroBarrier

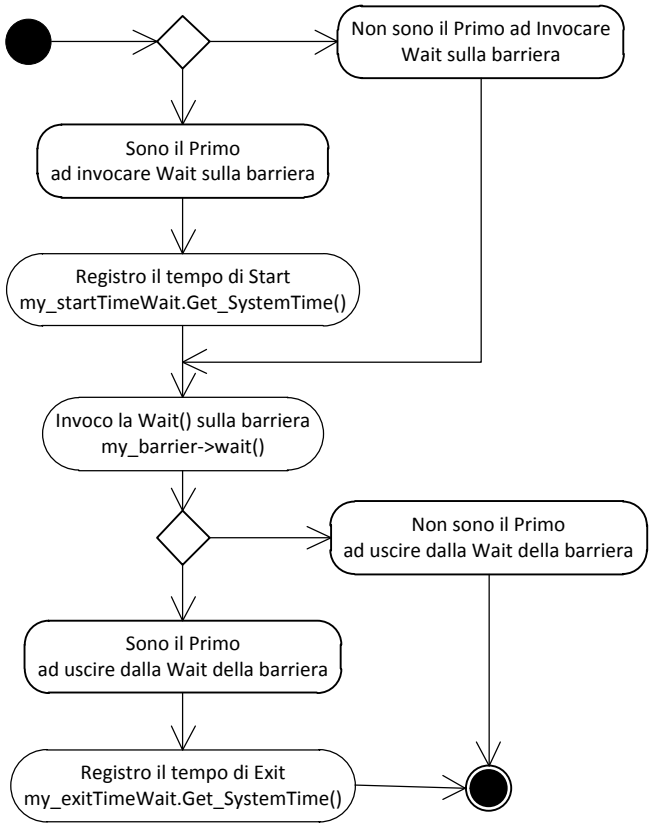
Si Implementa una struttura di sincronizzazione a barriera sulla base di quanto indicato nella Classe Interfaccia SyncroInterface. La struttura viene inizializzata per mezzo del costruttore:

```
SyncroBarrier my_sem(numBarrier);
```

dove dobbiamo specificare il numero di elementi **numBarrier** da attendere. Questa struttura è molto utile quando vogliamo sincronizzare o rispettare precedenze tra più Thread, infatti invocando la funzione Wait() il flusso di esecuzione si interrompe finché non vengono invocati numBarrier istruzioni di Wait() oppure diamo l'istruzione di sblocco globale Notify. Possiamo anche richiamare la funzione Wait(num) dove resettiamo la barriera impostando un nuovo numero di elementi di cui uno è già quello corrente.

Le funzioni membro della classe sono quindi:

- **Wait():** Funzione di Attesa della Sincronizzazione



- **Wait(num)**: Reset della Struttura e Impostazione di un nuovo numero di elementi di attendere
- **Notify():** Sblocco tutti i Thread in attesa Ereditando le proprietà della classe SyncroInterface possiamo anche recuperare l'istante di sblocco della barriera e il numero di elementi bloccati per mezzo della funzione:
- **Get_StartTime()**;
- **Get_NumElementBlocked()**;

5.4.3. SyncroSignal

La classe ci permette di sincronizzare due singoli eventi, deriva ancora dall'interfaccia SyncroInterface, ma per il suo sviluppo ci siamo basati sulla precedente classe dove abbiamo imposto il numero di elementi pari a 2.

Le funzioni membro quindi anche in questa occasione sono:

- Wait(): blocca il thread;
- Notify(): sblocca il thread;

5.4.4. SyncroSleep

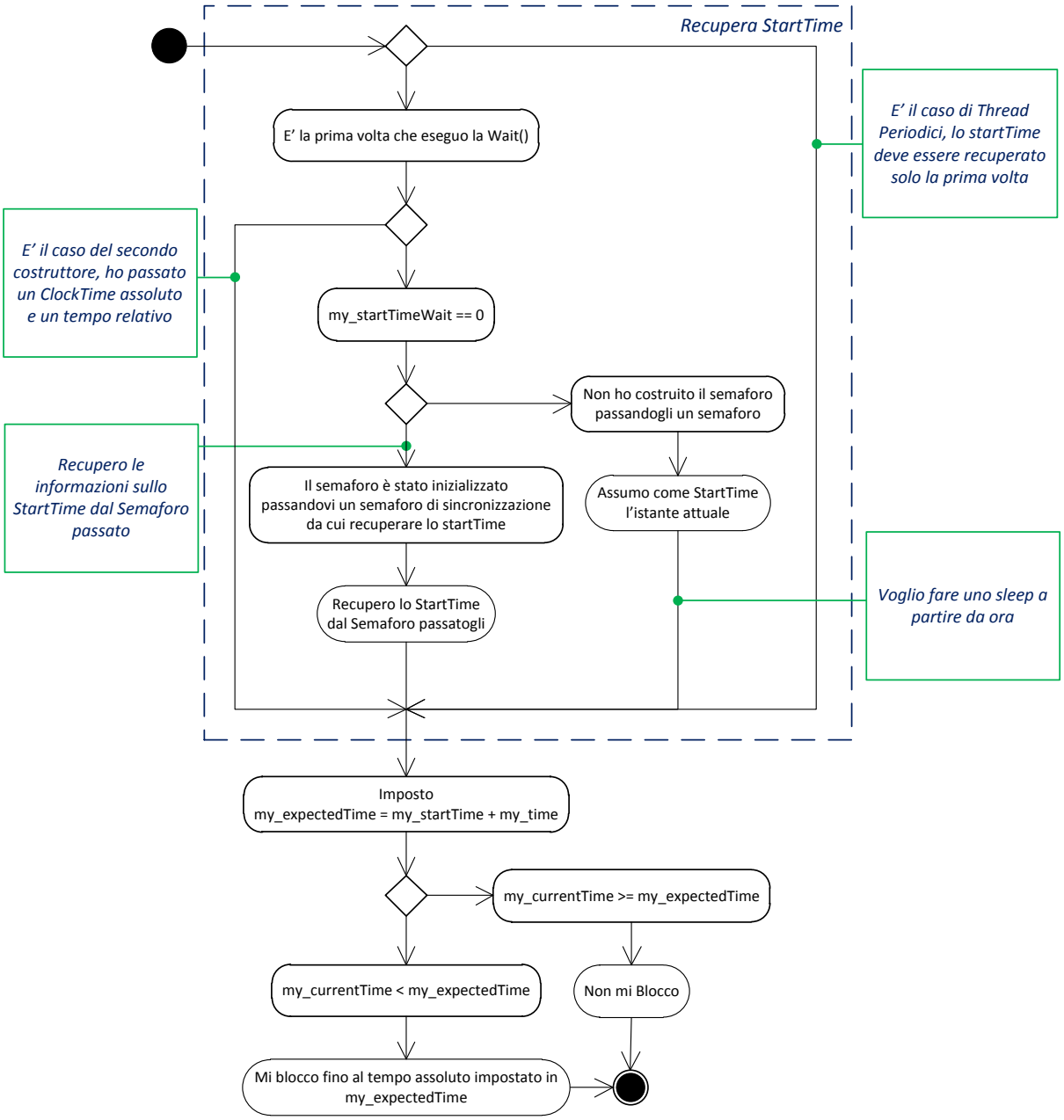
La classe ci permette di implementare una struttura che sospende il Thread in esecuzione fino ad un tempo assoluto indicato nel costruttore, abbiamo due modalità di costruzione:

```
SyncroSleep(long = DEFAULT_TIME_SLEEP, Syncro* = NULL);
SyncroSleep(ClockTime&, long);
```

Nel primo costruttore passiamo al SyncroSleep un tempo relativo di sospensione, il tempo assoluto a cui sommare questo valore lo ricaveremo dal semaforo Syncro passato come secondo parametro.

Nel secondo costruttore passiamo direttamente il tempo assoluto e il tempo relativo da sommarvi per lo sblocco.

Dopo che abbiamo costruito l'oggetto sarà sufficiente invocare la funzione membro Wait(). Analizziamone il funzionamento in quanto questa avrà comportamenti diversi a seconda del suo utilizzo:



5.4.5. Thread

La classe thread che abbiamo implementato è un'estensione della libreria boost::thread, a questa abbiamo aggiungo le possibilità di personalizzare la Priorità e la politica di Schedulazione, eseguire una funzione di inizializzazione che sarà opzionale, impostare un'eventuale Sincronizzazione di Start con altri Thread.

L'idea base è quella di chiamare la thread della boost non con la funzione che noi gli passiamo, ma con una funzione da noi creata appositamente interna alla classe che rispetta i nostri requisiti, vediamo quindi come questa è stata concepita:

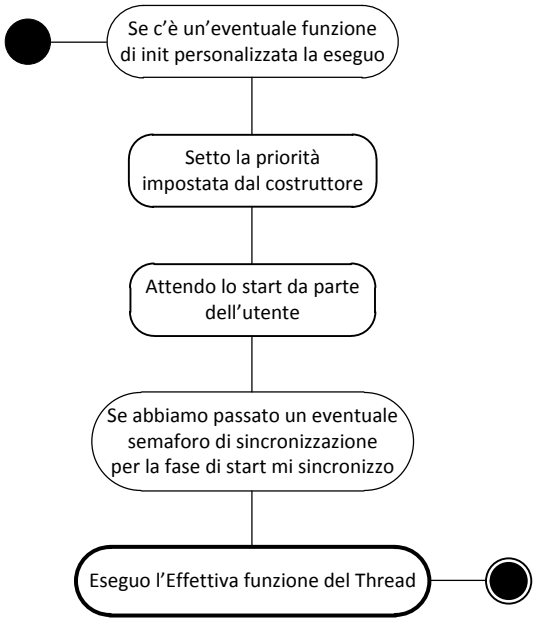


Figura 27. Funzione Membro Thread::Functor()

Come possiamo notare abbiamo arricchito la semplice funzione che passiamo al thread con una serie di utility. Tale funzione è inoltre dichiarata come virtuale in modo che se deriviamo la classe e ridefiniamo questa funzione, potremo mantenere tutte le funzionalità del thread relative ai cambi di priorità e politica di schedulazione, e ridefinire la funzione che esegue il thread stesso (è il principio che abbiamo adottato per creare le classi PeriodicRTThread e ModeThread che vedremo successivamente). E' importante però ricordare che se ridefiniamo la funzione virtuale Functor() dobbiamo inserirvi la funzione relativa al settaggio della priorità a meno che non si decida di eseguirla a parte.

Analizziamo i parametri che dobbiamo passare a un Thread quando lo dichiariamo:

```
Thread th( THREAD_FUNCTION,
           SyncroInterface*,
           PRIORITY,
           POLICY)
```

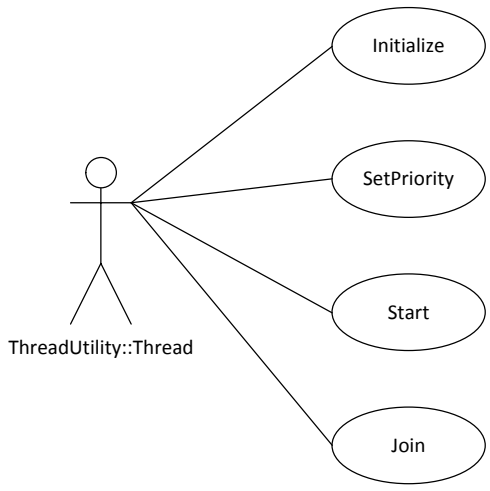
Parameters:

- **THREAD_FUNCTION:** è il puntatore a void della funzione passata al Thread. Grazie alla Libreria boost::bind che restituisce un puntatore a void senza parametri, possiamo passare al costruttore qualunque funzione con un numero arbitrario di parametri[8]
- **SyncroInterface&:** è il Semaforo impostato per l'eventuale sincronizzazione dello start del Thread
- **PRIORITY:** e la priorità di schedulazione vogliamo impostata per il Thread
- **POLICY:** è la Politica di schedulazione che vogliamo impostare per il Thread

Da tener presente che questa implementazione è utilizzabile sia sulle piattaforme basate su sistemi linux che sui sistemi windows, abbiamo cioè isolato ciò che diversificava le varie piattaforme per mezzo della classe **SchedParam** la cui trattazione è rimandata alla prossima sezione del capitolo.

Il costruttore si occupa solo della memorizzazione di tutti i parametri passati, per il settaggio dobbiamo chiamare la funzione Init() che carica il Thread vero e proprio. Il Thread Caricato chiama l'eventuale funzione di inizializzazione, setta la priorità e la politica di schedulazione e quindi si blocca. Per mandarlo in esecuzione sarà necessario invocare la funzione Start.

Detto questo vediamo i casi d'uso disponibili all'utente:



L'Attività Initialize ha il compito di memorizzare un'eventuale funzione di inizializzazione personalizzata, di creare il thread passandogli la Functor() e di inizializzare le variabili dipendenti dalla creazione del thread stesso.

[8] Se ad esempio vogliamo passare la funzioni:

```
void f1(int a, int b);
void f2(int a, int b, int c);
```

Potremo chiamare il costruttore in entrambi i casi con il costrutto:

```
Thread th1(boost::bind(f1, a, b);
Thread th2(boost::bind(f2, a, b, c);
```

L'Attività Start se non è stata eseguita la funzione di Inizializzazione la esegue e quindi sblocca il Thread che a questo punto è stato certamente creato.

L'Attività Join permette di attendere la terminazione del Thread.

L'Attività SetPriority imposta per il thread corrente la priorità e la politica passata o settata precedentemente dal costruttore.

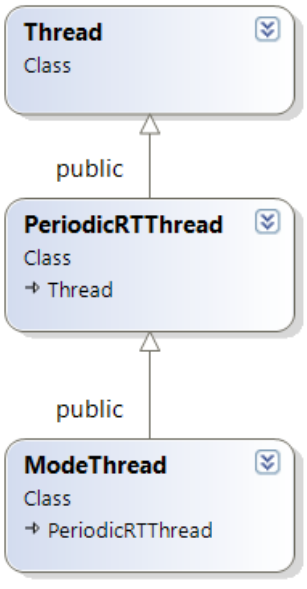
Le attività come accennato devono essere quindi eseguite con la seguente precedenza:

Questa è comunque garantita dalle singole funzioni.

La classe è quindi stata così sviluppata:

Thread
#my_thread : thread * #my_nativeHandle : native_handle_type #my_id : id #my_startTime : ClockTime #my_endTime : ClockTime #flag_init : bool #my_function : THREAD_FUNCTION #my_functionInit : THREAD_FUNCTION #my_barrierStart : barrier * #my_syncroStart : Syncro * #my_priority : PRIORITY #my_policy : POLICY
#Functor() +Thread(in function : THREAD_FUNCTION, in syncroStart : Syncro*, in priority : PRIORITY, in policy : POLICY) +~Thread() +Init(in my_init : THREAD_FUNCTION) +Start() +Join() +Set_Priority() : int +Set_Priority(in priority : PRIORITY) : int +Get_Id() : id & +Get_StartTime() : ClockTime & +Get_EndTime() : ClockTime & +Get_SchedPriority() : PRIORITY

Dalla Classe Thread abbiamo derivato la classe PeriodicRTThread e da quest'ultima la classe ModeThread.



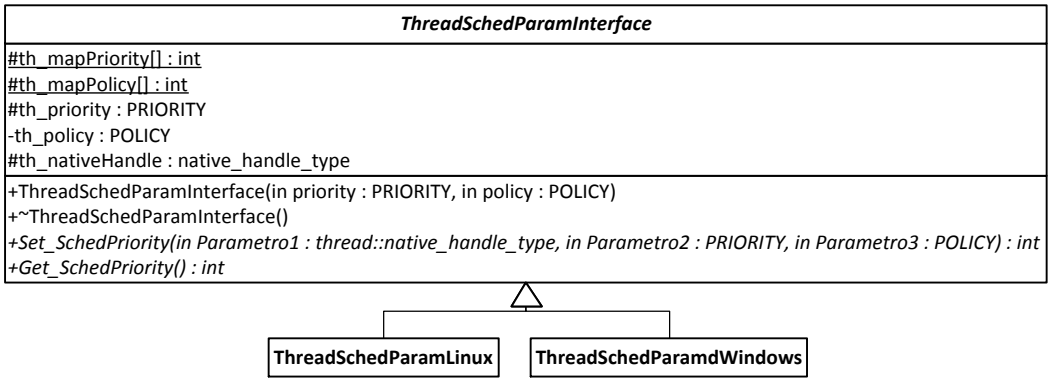
5.5. SchedParam

Un obiettivo che ci eravamo preposti per lo sviluppo della libreria era la trasportabilità sulle varie piattaforme. Per farlo abbiamo implementato una struttura separata dal Thread che ci permette di impostare agevolmente la proprietà di schedulazione sulle varie piattaforme. Abbiamo implementato un'interfaccia dalla quale derivare le istanze relative alle piattaforma utilizzata e nel file *ThreadSchedParam.h* imposteremo per mezzo di una macro la classe che verrà associata alla SchedParam.

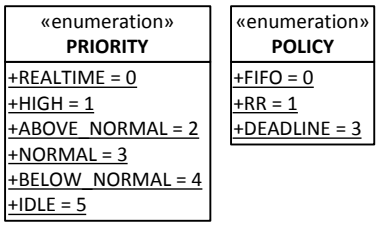
```
#ifdef WIN32
#include "ThreadSchedParamWindows.h"
typedef ThreadSchedParamWindows SchedParam;
#elif defined(LINUX)
#include "ThreadSchedParamLinux.h"
typedef ThreadSchedParamLinux SchedParam;
#endif
```

Nella nostra libreria abbiamo previsto l'utilizzo della piattaforma Windows e Linux. Nel caso di Linux abbiamo preso in considerazione un kernel modificato che estende le proprie funzionalità alle applicazioni Real-Time legate alla schedulazione EDF.

Quando quindi utilizziamo i Thread dovremo definire se ci troviamo su una piattaforma Linux (LINUX) o Windows (WIN32).



La classe Interfaccia presenta al suo interno la definizione di due array statici, questi indicano i valori di priorità (*th*_mapPriority) e politica (*th_mapPolicy*) di schedulazione sul sistema operativo che stiamo considerando. Per realizzare la trasportabilità del codice è quindi sufficiente derivare dall'interfaccia una classi per ogni sistema operativo considerato, inizializzare gli array statici e implementare le funzioni astratte dell'interfaccia. Le proprietà di schedulazione all'interno della mappa dovranno essere indirizzate per mezzo degli enumerati PRIORITY e POLICY.



Vediamo l'assegnazione della priorità nei due casi Windows

```
int ThreadSchedParamInterface::th_mapPriority[] = {
    THREAD_PRIORITY_TIME_CRITICAL,
    THREAD_PRIORITY_HIGHEST,
    THREAD_PRIORITY_ABOVE_NORMAL,
    THREAD_PRIORITY_NORMAL,
    THREAD_PRIORITY_BELOW_NORMAL,
    THREAD_PRIORITY_LOWEST
};
```

e Linux:

```
int ThreadSchedParamInterface::th_mapPriority[] = { 98, 80, 60, 50, 40, 20 };
```

La funzioni membro astratta che deve essere implementata per le varie piattaforme è:

```
Set_SchedPriority(
                    boost::thread::native_handle_type,
                    PRIORITY,
                    POLICY);
```

Per mezzo di questa struttura ci è stato quindi possibile trasportare la libreria sia su piattaforma Linux che Windows, e utilizzare su ognuna ciò che la piattaforma stessa prevede.

Nel caso Linux questa sceglierà tra le due soluzioni:

- **Set_SchedPriority_FIFO_RR**: che richiama le funzionalità FIFO e RR di schedulazione della pthread.
- **Set_SchedPriority_Deadline**: che richiama le funzionalità sviluppate nella dl_syscalls utilizzata nel kernel modificato di Linux che abbiamo utilizzato per le sperimentazioni (6) (7) (8).

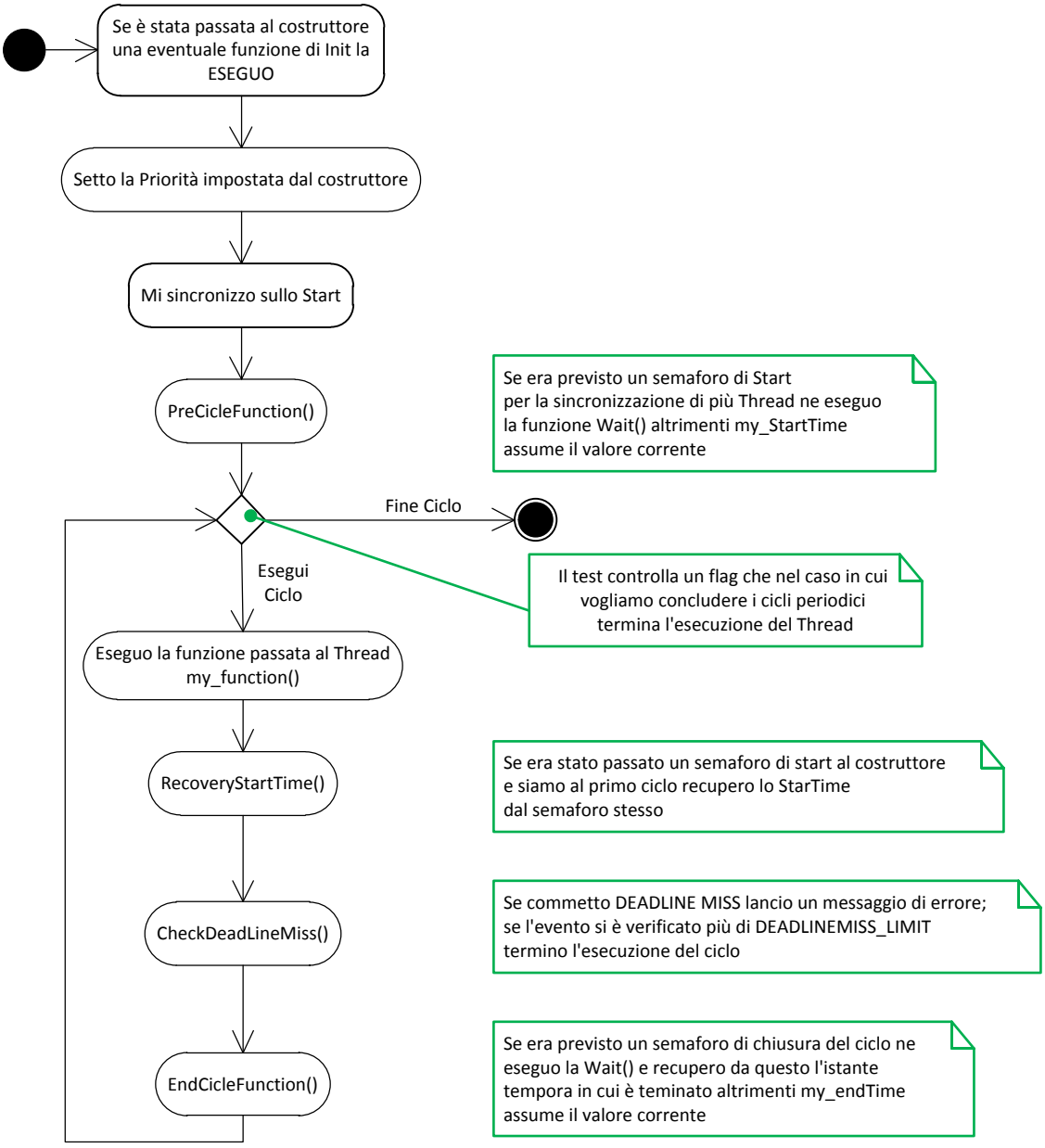
Nel caso Windows l'algoritmo di scheduling è di tipo prioritario a divisione di tempo con preemption: verranno sempre eseguiti i thread a priorità più alta. Su Windows un Thread viene eseguito fino a quando non è sottoposto a preemption da un thread a priorità più alta, termina o esaurisce il suo quanto di tempo. In Windows i Thread vengono suddivisi in classi prioritarie e ogni classe a sua volta possiede vari gradi di priorità. Per settare la schedulazione in questo caso dobbiamo quindi chiamare la **SetPriorityClass** e la **SetThreadPriority**.

5.6. PeriodicRTThread

Estensione della Libreria boost::thread che istanzia un Thread Periodico Real Time, il periodo è da considerarsi in microsecondi.

5.6.1. Funzione membro virtuale Functor()

La classe è stata derivata dalla Thread e di questa abbiamo ridefinito la funzione membro virtuale Functor(), in questo modo abbiamo mantenuto tutte le proprietà per la modifica delle proprietà di schedulazione. Come valore aggiunto, in questa classe è stata lasciata la possibilità di ridefinire le funzioni virtuali che troviamo nei vari punti strategici del ciclo periodico della Functor. Possiamo inoltre passare al Thread Periodico due semafori di sincronizzazione uno che verrà eseguito prima dello start del ciclo periodico e uno alla fine di ogni ciclo periodico.



L'inserimento dei Semafori di Start e di End ci permetterà per gli sviluppi futuri la

massima flessibilità in quanto potremmo utilizzare la struttura creata per realizzare ad esempio catene di Thread e Thread Aperiodici ecc.

5.6.2. Costruttore

```
PeriodicRTThread::PeriodicRTThread(
        THREAD_FUNCTION function,
        long period = PERIOD_DEFAULT,
        PRIORITY priority = DEFAULT_SCHED_PRIORITY,
        POLICY policy = DEFAULT_SCHED_POLICY,
        Syncro* syncroStart = NULL,
        Syncro* syncroEnd = NULL
)
```

Al costruttore vanno passati tutti i parametri necessari per il successivo settaggio:

- **THREAD_FUNCTION**: è la funzione che deve essere eseguita periodicamente
- **long period**: è il periodo da considerarsi in microsecondi con cui la funzione deve essere eseguita
- **PRIORITY**: è la priorità che verrà assegnata al thread per la schedulazione
- **POLICY**: è la politica e gli eventuali parametri che serviranno allo schedulatore per la relativa impostazione
- **syncroStart**: è il semaforo di cui, se presente, eseguiremo la Wait() prima di entrare nel ciclo periodico
- **syncroEnd**: è il semaforo di cui, se presente, eseguiremo la Wait() alla fine di ogni ciclo.

5.6.3. bool Join(int nCycles)

La funzione membro **Join**, dopo *nCycles* imposta il flag di terminazione del ciclo determinando la fine del Thread periodico.

5.6.4. Exit

Imposta il flag di terminazione del ciclo determinando la fine del Thread periodico.

5.7. Gestione dei Cambi di Modi

Prima di addentrarci sullo sviluppo del sistema riguardante i cambi di modo approfondiamone la trattazione che abbiamo introdotto in precedenza.

Abbiamo detto che prima, dopo e durante una transizione dobbiamo rispettare i requisiti di:

- Schedulabilità
- Periodicità
- Prontezza
- Consistenza delle Variabili

L'uso di adeguati offset ci aiuterà a realizzare questi requisiti. Faremo cioè ritardare l'attivazione dei Task del nuovo-modo. Se gli offset sono abbastanza grandi, l'intero carico del vecchio-modo sarà completato prima che i task del nuovo-modo vengano introdotti, ciò ci permette di eliminare il sovraccarico che avremmo nelle transizione. Anche il problema della consistenza può essere risolto introducendo gli offset, possiamo infatti ritardare i task del nuovo-modo fino a che i dati non hanno raggiunto la consistenza.

Sembrerebbe quindi che usando offset lunghi possiamo risolvere tutti i nostri problemi; nel caso di task immutati, romperemmo la loro periodicità, e offset grandi non sono proficui nemmeno per quanto riguarda la prontezza: il ritardo contraddice la prontezza.

Stiliamo quindi un metodo per la classificazione dei cambi di modo.

Ci sono tre caratteristiche principali dei protocolli in base alle quali riusciamo generare di criteri di classificazione (9):

i) *In base alla **Capacità di abbandonare i task del vecchio-modo** all'arrivo di una MCR possiamo distinguere i seguenti casi:*
   - *quando arriva una MCR, tutti i task del vecchio-modo vengono immediatamente abortiti*. Questo comportamento può essere adeguato nel caso di un settaggio di un modo *"emergency"*, permettendo che il nuovo modo cominci subito. Questo metodo però determina la perdita della consistenza dei dati;
   - *dopo l'arrivo di una MCR, tutti i task del vecchio-modo possono essere completati normalmente.* In questo modo eliminiamo il precedente svantaggio però possiamo introdurre una grande latenza determinando una scarsa prontezza;
   - *quando arriva una MCR, solo alcuni task vendono abortiti.* Questa opzione è più flessibile, però richiede un trattamento diverso per i task

del vecchio-modo che sono in esecuzione nella fase di transizione: un task potrà essere o non potrà essere abortibile a seconda del modo che verrà impostato.

ii) *In base al* ***Metodo di attivazione dei task che rimangono immutati*** *nella transizione possono presentarsi i seguenti casi:*
   - *protocolli con periodicità:* nell'ambito di questi protocolli i task immutati vengono eseguiti indipendentemente dal cambio di modo in corso. Essi conservano cioè il proprio passo di attivazione;
   - *protocolli senza periodicità:* l'attivazione dei task immutati può essere ritardata, il passo di attivazione può quindi essere influenzato dalla transizione. La perdita della periodicità a volte può risultare necessario per mantenere la fattibilità della transizione e per garantire la consistenza.

iii) *In base alla* ***Capacità del protocollo di unire l'esecuzione dei task del nuovo-modo a quelli del vecchio-modo*** *durante la transizione possiamo distinguere:*
   - *protocolli sincroni:* dove i task del nuovo-modo non sono mai rilasciati fino a quando tutti i task del vecchio-modo non completano la loro ultima attivazione. I protocolli sincroni non richiedono una nuova analisi di schedulabilità, poiché il sovraccarico del cambio di modo non accade, lo stato stabile di schedulabilità è quindi sufficiente affinché il cambio di modo sia schedulabile. Esistono protocolli sincroni con e senza periodicità.
   - *protocolli asincroni:* dove durante la transizione è consentito eseguire una combinazione dei task del nuovo-modo e del vecchio. Questi protocolli richiedono un'analisi specifica di scedulabilità poiché la quota di lavoro durante la transizione può essere superiore a quello dello stato di stabilità.

5.7.1. Modello del protocollo implementato: mininum single offset: con periodicità

Alla luce di questo metodo di classificazione passiamo a caratterizzare la nostra scelta implementativa. Il protocollo da noi sviluppato rispecchia le seguenti caratteristiche:

- è di tipo sincrono,
- mantiene l'attivazione periodica dei task che rimangono immutati durante un cambio di modo,
- i task che erano attivi nel vecchio-modo ma non nel nuovo-modo potranno terminare la propria esecuzione prima di attivare quelli nuovi, non verranno cioè abortiti.

Questa scelta implementativa, come già detto, non necessità di un'analisi approfondita sulla schedulazione in quanto durante la fase di transizione non avremo sovraccarico.

Il nostro protocollo è conosciuto come ***minimum single offset (without periodicity)*** ed

è stato presentato in Real(2000).

La tipologia dei protocolli a singolo offset prevede di far ritardare tutti i task del nuovo-modo fino ad un determinato istante dopo l'arrivo della MCR, l'offset utilizzato sarà lo stesso per tutti i task del nuovo-modo.

Secondo questo protocollo quando arriva una MCR al sistema, i task del vecchio-modo possono essere completati se sono attualmente attivi, ma questi non dovranno avere un'ulteriore esecuzione. I task che rimangono immutati e che saranno attivi anche nel nuovo-modo non dovranno subire modifiche sulla loro attivazione. Il singolo offset Y applicato a tutti i task del nuovo-modo può quindi essere calcolato come:

$$Y = \sum_{j \in Old} C_j + \sum_{j \in Unch} \left\lceil \frac{Y}{T_j} \right\rceil C_j$$

Il secondo termine di questa equazione rappresenta l'interferenza dei successivi rilasci dei thread immutati durante la transizione. Nella seguente figura possiamo vedere un possibile esempio di esecuzione nell'ambito del nostro protocollo. Lo svantaggio di questo protocollo riguarda la prontezza, l'offset può infatti essere anche molto grande nel caso in cui un Task debba concludere la propria esecuzione e il suo periodo di attivazione sia grande. Dobbiamo inoltre affrontare il problema della consistenza in quanto per un certo intervallo di tempo, task del nuovo-modo e del vecchio possono funzionare simultaneamente, l'esecuzione dei task immutati deve continuare la propria attivazione periodica dal vecchio al nuovo modo.

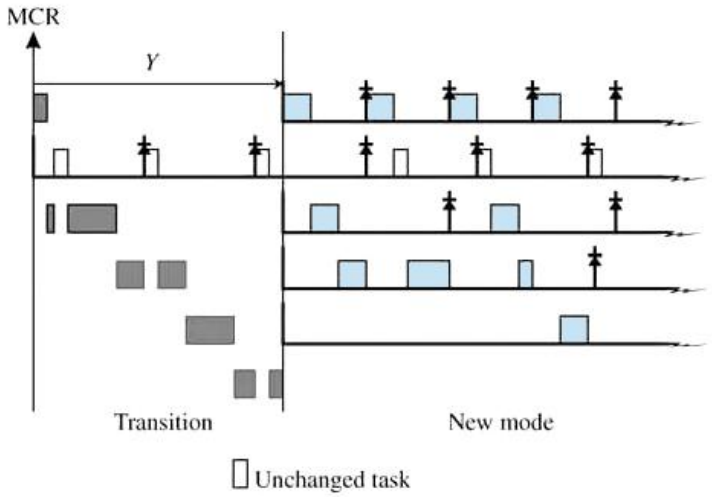
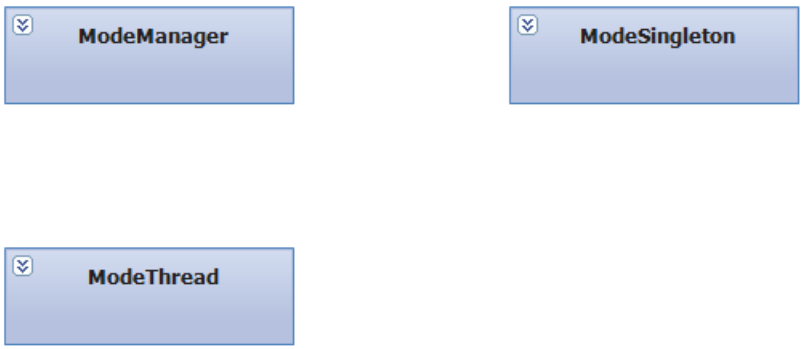


Figura 28. Minimum Single Offset (Without Periodicity)

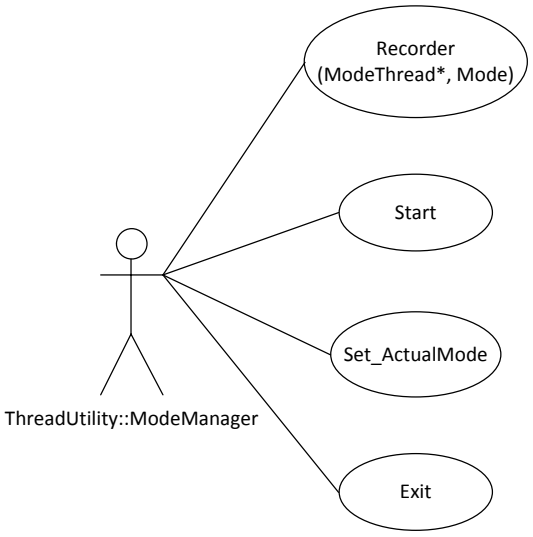
5.7.2. Modellazione Software: Suddivisione dei Compiti

Nella nostra soluzione, l'implementazione e la gestione dei modi viene suddivisa su tre componenti principali:



- **Manager dei Task**: si occupa del settaggio dei modi, imposta il modo attuale del sistema e gestisce la transizione tra i due moti. E' su questa struttura che i Thread dovranno registrarsi indicando su quali modi saranno attivi, sarà poi il ModeManager che aggiornerà la struttura ModeSingleton
- **ModesSingleton**: è la struttura che contiene tutte le informazioni necessarie per la gestione dei cambi di modo, le associazioni tra i thread e i modi relativi, i semafori su cui i thread si dovranno interrompere, il modo attuale e quello precedente. Sono anche presenti tutte le funzioni necessarie per la gestione e il controllo dell'attivazione dei Thread sul modo attuale, l'attivazione di tutti i thread e il settaggio del modo attuale.
- **Mode Thread**: questa struttura attiva un Thread Periodico che esegue la funzione che noi gli passiamo e che viene impostato con le priorità, la politica e i settaggi da noi indicati.

5.7.3. Casi d'Uso – ModeManager



Come possiamo notare, nella nostra struttura una volta creato un ModeThread, sarà sufficiente che il master lo registri associandovi i modi su cui questo dovrà essere attivo. Quando tutti i ModeThread saranno stati creati e registrati il Manager potrà

startare il sistema, che come default partirà dallo stato di riposo (STANDBY). A questo punto il Manager potrà settare un nuovo modo per mezzo della Set_ActualMode e in automatico i ModeThread su questo attivi si sveglieranno. Quando vogliamo concludere l'esecuzione sarà sufficiente invocare la Exit.

Dal diagramma di sequenza possiamo notare appunto la successione delle azioni che vengono svolte dal sistema nel processo del cambio dei Modi.

Vediamo un esempio di uso in cui dichiariamo 3 Thread e impostiamo i seguenti Modi di attivazione:

- TH1 = UNO
- TH2 = UNO, DUE
- TH3 = UNO, DUE, TRE

Dovremo scrivere il seguente codice:

```
ModeThread th1(funzione1)
ModeThread th2(funzione2)
ModeThread th3(funzione3)
```

Dichiariamo la Struttura ModeManager:

```
ModeManager my_modeManager;
```

Impostiamo i modi relativi:

```
my_modeManager.Recorder(&th1, UNO);
my_modeManager.Recorder(&th2, UNO);
my_modeManager.Recorder(&th2, DUE);
my_modeManager.Recorder(&th3, UNO);
my_modeManager.Recorder(&th3, DUE);
my_modeManager.Recorder(&th3, TRE);
```

Starto il sistema:

```
my_modeManager.Start();
```

Imposto il Mode Attuale:

```
my_modeManager.Set_ActualMode(DUE);
```

Andranno in esecuzione TH2 e TH3, potremo terminare l'esecuzione con il comando:

```
my_modeManager.Exit();
```

Analizziamo ora nello specifico come sono costituite le tre classi Modes, ModeThread visto che la ModeManager semplicemente richiama le funzioni della ModesSingleton.

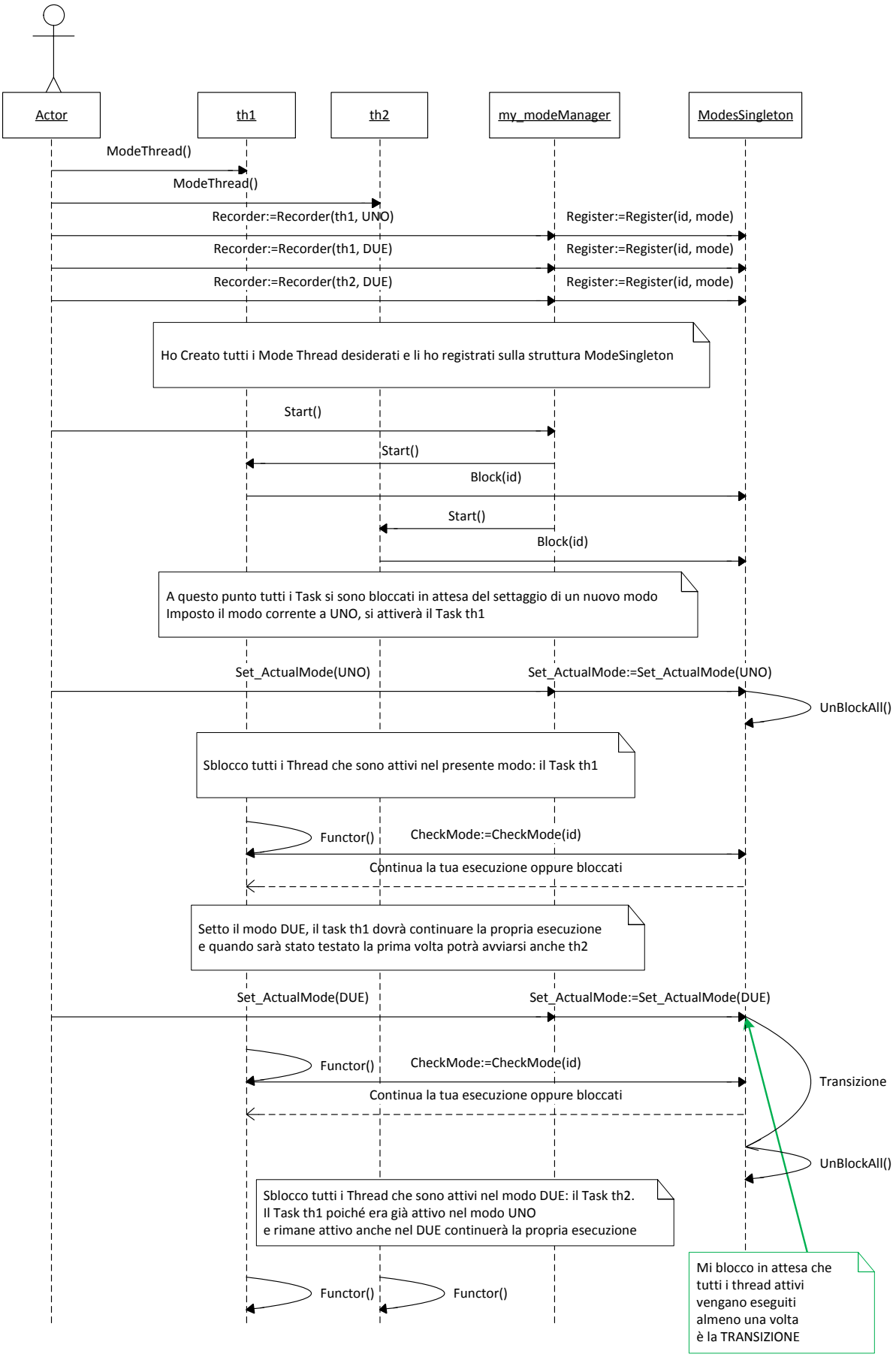
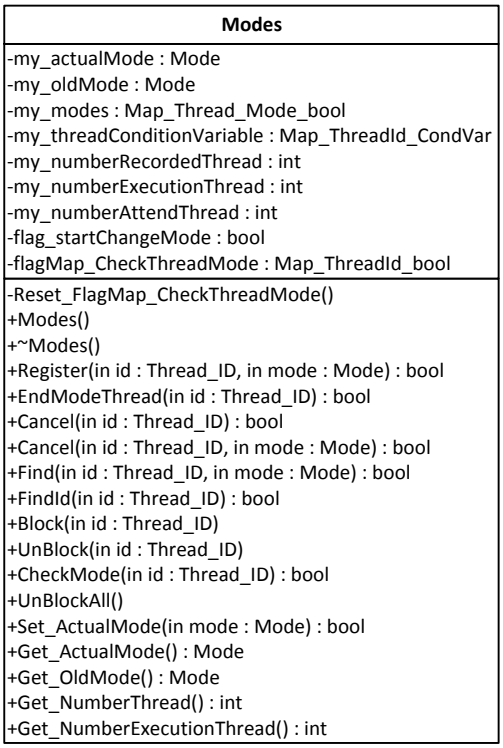


Figura 29. Diagramma temporale di un esempio di applicazione di Cambi di Modo

5.7.4. Modes e ModesSingleton

La classe ModesSingleton è l'implementazione Singleton della classe Modes, essendo Singleton la struttura Modes sarà unica per tutta l'istanza del processo (5).

Questa struttura implementa una classe che si occupa della memorizzazione delle informazioni per il cambio di modo dei Thread Periodici Real-Time.



Nello specifico per ogni Thread Periodico Real-Time, memorizziamo in una mappa bidimensionale *my_modes* l'Id e i vari Modi per cui il Thread deve essere attivato. La struttura *my_threadConditionVariable* memorizza un semaforo condizionale per ogni Thread registrato sulla struttura Modes. Questo si occuperà di bloccare i Thread a seconda che debbano essere eseguiti oppure no.

Ogni ModeThread può Registrare o Cancellare nella struttura i propri Modi, potremo inoltre Cancellare globalmente tutte le funzioni relative a un Thread_ID:

- **Register**(Thread_ID, Mode)
- **Cancel**(Thread_ID)
- **Cancel**(Thread_ID, Mode)

Possiamo verificare la presenza di una registrazione per mezzo delle funzioni di ricerca:

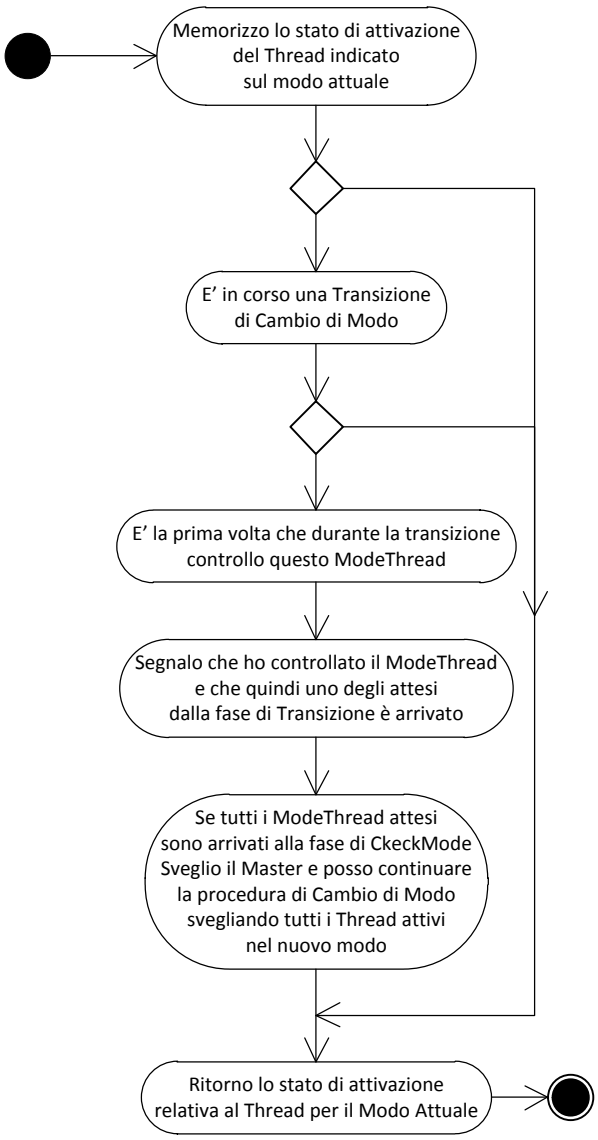
- **FindId**(Thread_ID)
- **Find**(Thread_ID, Mode)

La Funzione **CheckMode**() verifica se il Thread indicato è abilitato oppure no al corrente Modo settato. La Funzione **UnBlockAll**() sveglia tutti i Thread che non erano attivi nel vecchio modo ma lo sono attualmente, chiaramente nel caso di Modo

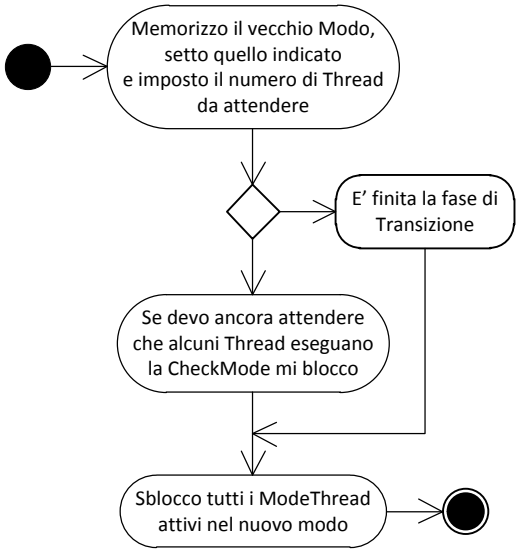
ON_ALL li sveglia tutti indiscriminatamente. La Funzione **Set_ActualMode**(Mode) imposta il Modo corrente a quello indicato salvando quello vecchio, attende la terminazione dei Thread che erano attivi nel vecchio modo ma non lo sono più in quello Attuale, quindi chiama la UnBlockAll() per attivare tutti i Thread che con il modo attuale devono essere attivi.

Sono degne di nota le funzioni membro CheckMode e Set_ActualMode:

CheckMode:



Set_ActualMode:



5.7.5. ModeThread

La classe definisce un tipo Thread Periodico RealTime dove in più sono state sviluppate le funzionalità per la gestione dei cambi di modo (è derivata dalla classe PeriodicRTThread).

Il Thread viene creato associandovi la funzione da mandare in esecuzione periodicamente, poi viene bloccato in attesa del segnale di Start. La funzione di Exit permette di interrompere l'esecuzione del Thread. Oltre alla funzione da eseguire periodicamente potremo passare al Mode Thread anche una funzione di inizializzazione, entrambe dovranno essere però di tipo boost::function (THREAD_FUNCTION). Per ogni PeriodicRTThread potremo settare inoltre Periodo, Priorità, Politica e semaforo di sincronizzazione iniziale e finale.

Vediamo quindi le varie fasi per una corretta esecuzione:

- Chiamata del **Costruttore**:

```
ModeThread::ModeThread  (
              THREAD_FUNCTION function,
              long period = MODE_PERIOD_DEFAULT,
              PRIORITY priority = MODE_PRIORITY_DEFAULT,
              POLICY policy = MODE_POLICY_DEFAULT,
              THREAD_FUNCTION initFunction = NULL,
              Syncro* syncroStart = NULL,
              Syncro* syncroEnd = NULL )
```

Il Costruttore oltre all'inizializzazione delle variabili si occupa anche dell'inizializzazione e della creazione vera e propria del Thread che però per essere eseguito ha bisogno del nostro Start(). La chiamata può essere fatta in diversi modi a seconda delle nostre esigenze:

```
ModeThread th(funzioneThread);
```

```
ModeThread th(funzioneThread, periodo);
ModeThread th(funzioneThread, periodo, priorità);
ModeThread th(funzioneThread, periodo, priorità, InitFunction);
ModeThread th(funzioneThread, periodo, priorità, InitFunction, SemaforoStart);
ModeThread th(funzioneThread, periodo, priorità, InitFun, SemStart, SemEnd);
```

Parametri:

THREAD_FUNCTION:	Funzione che deve essere eseguita periodicamente
long:	è il periodo di ripetizione della funzione periodica
PRIORITY:	è la priorità che viene assegnata al Thread
THREAD_FUNCTION:	E' la funzione che può essere passata al Thread e che viene eseguita prima dell'inizio della periodicità
SyncroStart*:	E' il semaforo di sincronizzazione per lo Start del Thread
SyncroEnd*:	E' il semaforo di sincronizzazione alla fine di ogni Ciclo

- **Start** del Thread:

```
th.Start();
```

--------- *Registrazione e Settaggio dei Modi gestito dal Mode Manager* ----------

- Quando è in esecuzione posso chiamare la terminazione del processo:

```
th.Exit();
```

La classe deriva direttamente dalla PeriodicRTThread e quindi è stato sufficiente ridefinire le funzioni di pre-ciclo, fine-ciclo e uscita-ciclo.

PreCicleFunction:

Definisce le operazioni da eseguire prima di iniziare l'esecuzione Periodica:

- se presente il semaforo di sincronizzazione di Start mi sincronizzo su quello
- mi blocco in attesa che il ModeManager mi sblocchi mandandomi in esecuzione
- quando il ModeManager mi sblocca setto il tempo di StartTime

EndCicleFunction:

Definisce le operazioni da eseguire alla fine di ogni ciclo: nel nostro caso dovremo controllare se il processo in esecuzione è abilitato per il Modo Corrente, in caso affermativo dovremo continuare l'esecuzione periodica, in caso il processo non sia più abilitato dovremo bloccarci in attesa che il Mode Manager ci risvegli.

ExitCicleFunction:

Sono le operazioni da eseguire quando l'esecuzione periodica viene interrotta: nel nostro caso, poiché il Task termina, dovremo semplicemente cancellarlo dalla struttura del ModeSingleton.

Cap. 6 - Esperimenti

Per dimostrare la consistenza e l'utilità della libreria sviluppata, abbiamo deciso di eseguire due esperimenti: il primo che valuta un confronto sulla tempistica di esecuzione della PeriodicRTThread rispetto alla medesima funzione eseguita senza di essa. Il secondo esperimento mostra la traccia del kernel lasciata dall'esecuzione di un programma che esegue i cambi di modi. La sperimentazione si è svolta su di una piattaforma Linux sperimentale dove il Kernel è stato modificato in modo da adattarsi ai sistemi real-time (6) (7). Tutti i dati ricavati sono stati elaborati mediante l'utilizzo delle funzionalità offerte dall'applicativo KernelShark che verrà illustrato nell'appendice (10) (11).

Abbiamo verificato gli esperimenti e le fasi di test anche sulla piattaforma Windows e anche se non disponiamo delle utility di Linux abbiamo riscontrato la correttezza dei risultati ottenuti per mezzo dell'utility TraceString da noi sviluppata.

6.1. Confronto tra PeriodicRTThread e pthread

In questa prima fase di sperimentazione vogliamo valutare le prestazioni del componente base della nostra libreria: la struttura che gestisce, imposta e manda in esecuzione un thread periodico. Abbiamo quindi sviluppato una funzione di test che manda in esecuzione 5 thread che eseguono per un tempo indicato da wcet e si ripetono in modo periodico come indicato. L'applicazione è stata prima sviluppata senza la nostra libreria, successivamente utilizzandola.

La funzione che verrà eseguita dai Thread periodici è la seguente:

```
void
ffunction(TraceString* tmp_trace, string str, long tmp)
{
  ClockTime startTime, time;
  startTime.Get_SystemTime();

  startTime += tmp;
  int i = 0;
  bool flag_end = false;

  while(!flag_end)
  {
    i++;
    time.Get_SystemTime();
    if(time >= startTime) flag_end = true;
  }
  (*tmp_trace) << str;
}
```

Questa funzione semplicemente rimane in esecuzione occupando le risorse del kernel per un tempo indicato da tmp (rimango bloccato nel ciclo while per tmp

microsecondi).

Se vogliamo scrivere il codice senza la nostra libreria, dovremo riscrivere le funzioni che:

- settano la priorità e la politica di schedulazione del Thread (*SetSchedFifo e SetSchedDeadline*)
- realizzano il funzionamento periodico della funzione che noi vogliamo eseguire (*FunctionPeriodic*)

```
struct timespec
usec_to_timespec(unsigned long usec)
{
   struct timespec ts;
   ts.tv_sec = usec / 1000000;
   ts.tv_nsec = (usec % 1000000) * 1000;
   return ts;
}

void
SetSchedFIFO(boost::thread::native_handle_type native)
{
   int th_policy = SCHED_FIFO;
   int th_priority = 98;
   sched_param th_param;
   th_param.__sched_priority = th_priority;
   boost::thread::native_handle_type th_nativeHandle = native;
   pthread_t threadID = (pthread_t) th_nativeHandle;

   int retval = pthread_setschedparam(threadID, th_policy, &th_param);
   if(retval != 0) {
       cout << "warning!!!" << endl;
       exit(-1);
   }
}

void
SetSchedDeadline(long wcet, long periodo)
{
   struct sched_param_ex dl_params;

   memset(&dl_params, 0, sizeof(dl_params));
   dl_params.sched_priority = 0;
   dl_params.sched_runtime = usec_to_timespec(wcet);   // quanto tempo eseguire
   dl_params.sched_deadline = usec_to_timespec(wcet);  // quanto tempo eseguire
   dl_params.sched_period = usec_to_timespec(periodo); // periodico
   dl_params.sched_flags = SF_BWRECL_DL;
   int retval;
   retval = sched_setscheduler_ex((pid_t)syscall(SYS_gettid),
                                  SCHED_DEADLINE,
                                  sizeof(struct sched_param_ex),
                                  &dl_params);
```

```
    if(retval != 0) {
            cout << "warning!!!" << endl;
            exit(-1);
    }
}

void function(long wcet, boost::xtime startTime)
{
    boost::xtime tmp, time;
    tmp = startTime;
    tmp.nsec = tmp.nsec + wcet * 1000;
    if (tmp.nsec > 1000000000)
    {
        tmp.sec += tmp.nsec / 1000000000;
        tmp.nsec = tmp.nsec % 1000000000;
    }
    int i = 0;
    bool flag_end = false;

    while(!flag_end)
    {
        i++;
        boost::xtime_get(&time, boost::TIME_UTC);
        if((boost::xtime_cmp(time, tmp) >= 0) ? 1 : 0) flag_end = true;
    }
}

void
FunctionPeriodic(string str,
                    long wcet,
                    long periodo,
                    boost::barrier* barriera)
{
    barriera->wait();
    boost::xtime startTime,tmp;

    while(1)
    {
        boost::xtime_get(&startTime, boost::TIME_UTC);
        function(wcet, startTime);

        tmp = startTime;
        tmp.nsec = tmp.nsec + periodo * 1000;
        if (tmp.nsec > 1000000000)
        {
            tmp.sec += tmp.nsec / 1000000000;
            tmp.nsec = tmp.nsec % 1000000000;
        }
        boost::thread::sleep(tmp);
    }
}
```

```
int main() {
   long wcet = 10000;
   long periodo = 1000000;

   boost::barrier sem_barrier(6);

   boost::thread th1(boost::bind(FunctionPeriodic, "A", wcet, periodo, &sem_barrier));
   boost::thread th2(boost::bind(FunctionPeriodic, "B", wcet*2, periodo, &sem_barrier));
   boost::thread th3(boost::bind(FunctionPeriodic, "C", wcet*3, periodo, &sem_barrier));
   boost::thread th4(boost::bind(FunctionPeriodic, "D", wcet*4, periodo, &sem_barrier));
   boost::thread th5(boost::bind(FunctionPeriodic, "E", wcet*5, periodo, &sem_barrier));

   SetSchedFIFO(th1.native_handle());
   SetSchedFIFO(th2.native_handle());
   SetSchedFIFO(th3.native_handle());
   SetSchedFIFO(th4.native_handle());
   SetSchedFIFO(th5.native_handle());

   sem_barrier.wait();

   sleep(5);
   th1.interrupt();
   th2.interrupt();
   th3.interrupt();
   th4.interrupt();
   th5.interrupt();

   return 0;
}
```

Utilizzando la libreria da noi sviluppata semplifichiamo notevolmente il codice, basterà scrivere il main in quanto le utility di impostazione sono già tutte scritte. Dovremo unicamente scrivere:

```
int main() {

   long wcet = 10000;
   long periodo = 100000;

   TraceString trace;

   SyncroBarrier sem_start(5);
   SyncroSleep sem1(periodo, &sem_start);
   SyncroSleep sem2(periodo, &sem_start);
   SyncroSleep sem3(periodo, &sem_start);
   SyncroSleep sem4(periodo, &sem_start);
   SyncroSleep sem5(periodo, &sem_start);

   PeriodicRTThread th1(boost::bind(ffunction, &trace, "A", wcet),
                        periodo,
                        REALTIME, POLICY(FIFO, wcet, periodo),
                        &sem_start, &sem1);
```

```
    PeriodicRTThread th2(boost::bind(ffunction, &trace, "B", wcet*2),
                         periodo,
                         REALTIME, POLICY(FIFO, wcet*2, periodo),
                         &sem_start, &sem2);
    PeriodicRTThread th3(boost::bind(ffunction, &trace, "C", wcet*3),
                         periodo,
                         REALTIME, POLICY(FIFO, wcet*3, periodo),
                         &sem_start, &sem3);
    PeriodicRTThread th4(boost::bind(ffunction, &trace, "D", wcet*4),
                         periodo,
                         REALTIME, POLICY(FIFO, wcet*4, periodo),
                         &sem_start, &sem4);
    PeriodicRTThread th5(boost::bind(ffunction, &trace, "E", wcet*5),
                         periodo,
                         REALTIME, POLICY(FIFO, wcet*5, periodo),
                         &sem_start, &sem5);
    th1.Start();
    th2.Start();
    th3.Start();
    th4.Start();
    th5.Start();

    SLEEP(5);

    th1.Exit();
    th2.Exit();
    th3.Exit();
    th4.Exit();
    th5.Exit();

    cout << "TRACCIA: " << endl;
    cout << trace.GetStream() << endl;

    return 0;
}
```

Questa scrittura può essere ulteriormente semplificata creando una classe ad ok per la nostra applicazione dando una specializzazione della classe PeriodicRTThread:

```
#include "PeriodicRTThread.h"

class PeriodicThread {
  PeriodicRTThread* my_thread;
  SyncroSleep* my_syncroEnd;
public:
  PeriodicThread(THREAD_FUNCTION,
       long = PERIOD_DEFAULT,
       PRIORITY = DEFAULT_SCHED_PRIORITY,
       POLICY = DEFAULT_SCHED_POLICY,
       Syncro* = NULL);
  virtual ~PeriodicThread();
  void Exit();
};
```

```
PeriodicThread::PeriodicThread(THREAD_FUNCTION function,
                               long period,
                               PRIORITY priority, POLICY policy,
                               Syncro* syncroStart)
{
   if(syncroStart != NULL)
      my_syncroEnd = new SyncroSleep(period, syncroStart);
   else
   {
      ClockTime startTime;
      startTime.Get_SystemTime();
      my_syncroEnd = new SyncroSleep(startTime, period);
   }
   my_thread = new PeriodicRTThread(function, period,
                                    priority, policy,
                                    syncroStart, my_syncroEnd);
   my_thread->Start();
}

PeriodicThread::~PeriodicThread() {
   delete my_thread;
   delete my_syncroEnd;
}

void
PeriodicThread::Exit(){
   my_thread->Exit();
}
```

Come possiamo notare la scrittura del main si semplifica ulteriormente:

```
int main() {
   long wcet = 10000;
   long periodo = 100000;
   TraceString trace;

   SyncroBarrier sem_start(5);

   PeriodicThread th1(boost::bind(ffunction, &trace, "A", wcet),
                              periodo, REALTIME, POLICY(FIFO), &sem_start);
   PeriodicThread th2(boost::bind(ffunction, &trace, "B", wcet*2),
                              periodo, REALTIME, POLICY(FIFO), &sem_start);
   PeriodicThread th3(boost::bind(ffunction, &trace, "C", wcet*3),
                              periodo, REALTIME, POLICY(FIFO), &sem_start);
   PeriodicThread th4(boost::bind(ffunction, &trace, "D", wcet*4),
                              periodo, REALTIME, POLICY(FIFO), &sem_start);
   PeriodicThread th5(boost::bind(ffunction, &trace, "E", wcet*5),
                              periodo, REALTIME, POLICY(FIFO), &sem_start);
   SLEEP(5);
   th1.Exit();
   th2.Exit();
   th3.Exit();
   th4.Exit();
   th5.Exit();
```

```
    cout << "TRACCIA: " << endl;
    cout << trace.GetStream() << endl;
    return 0;
}
```

In questo modo abbiamo inoltre nascosto anche l'utilizzo di tutte le funzionalità interne della PeriodicRTThread.

Analizziamo ora le prestazioni per mezzo dell'utility KernelShark trattata nell'appendice. Questa ci permette di visualizzare la traccia delle applicazioni che vanno in esecuzione sul kernel.

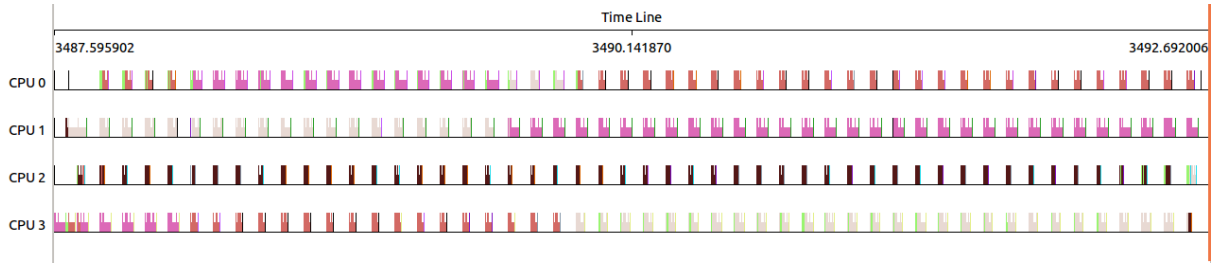


Figura 30. Traccia del Kernel durante l'esecuzione dell'applicazione

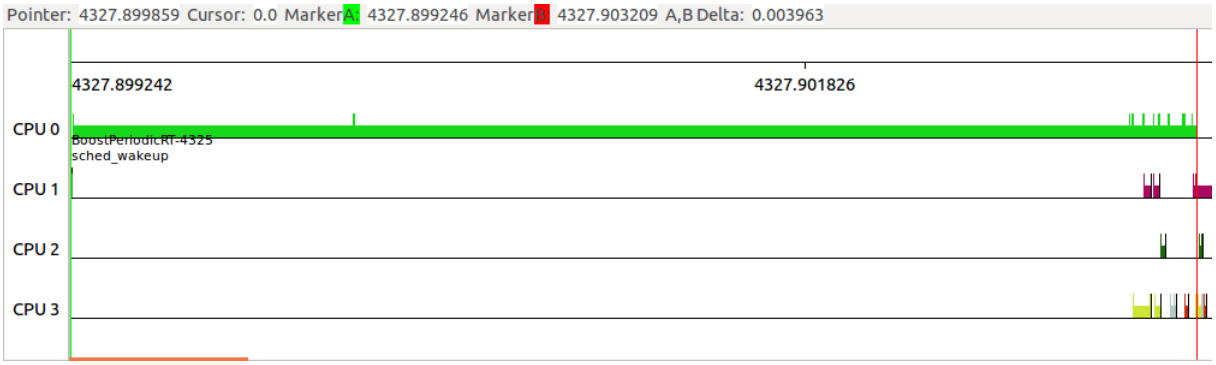


Figura 31. Traccia del Kernel in relazione all'applicazione senza la libreria sviluppata (dettaglio del Tempo di Setup iniziale)

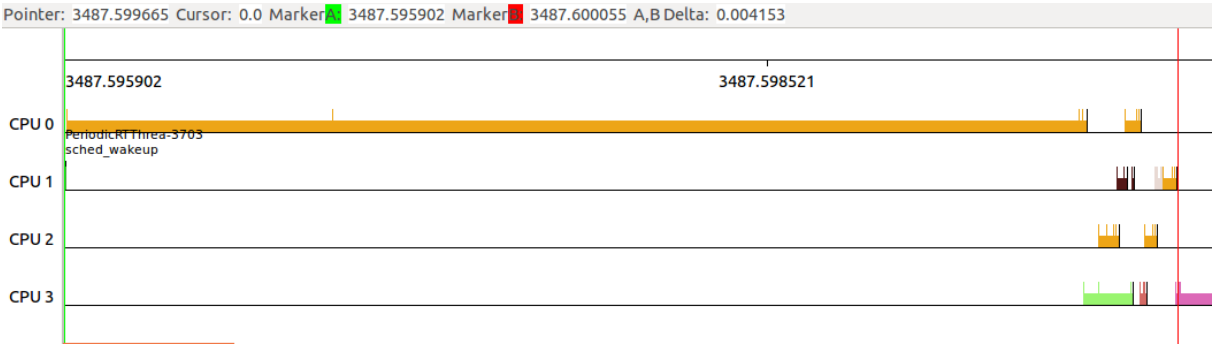


Figura 32. Traccia del Kernel in relazione all'applicazione della libreria utilizzata (dettaglio del Tempo di Setup iniziale)

Il primo grafico rappresenta l'esecuzione dell'applicazione a regime, e questa si è dimostrata uguale in entrambi i casi. Dal secondo e dal terzo grafico possiamo invece vedere il dettaglio del tempo di setup del sistema prima dello startup vero e proprio dei Thread. Nel caso in cui usiamo la pthread abbiamo un tempo di setup pari a 3,963ms, nel caso in cui usiamo la PeriodicRTThread invece il tempo di setup è pari a 4,153ms (come indicato dalla delta in alto ad ogni grafico); possiamo quindi affermare

che il tempo di setup iniziale è pressoché identico. Era questo l'obiettivo che ci eravamo prefissati: ottenere una struttura che semplificasse utilizzo dei Thread Periodici e che non causasse un decadimento delle prestazioni del sistema rispetto all'utilizzo classico.

6.2. Traccia del Kernel relativa al Cambio di Modo di Lavoro

Per il secondo esperimento svolto abbiamo voluto analizzare sempre con l'ausilio del KernelShark l'effettivo funzionamento di un'applicazione che imposta un set di thread, carica i modi in cui questi devono attivarsi e prova a cambiare il modo in cui il sistema si trova. Questo è il codice:

```
void
function (TraceString* tmp_trace, string str, long tmp)
{
   ClockTime startTime, time;
   startTime.Get_SystemTime();

   startTime += tmp;
   int i = 0;
   bool flag_end = false;

   while(!flag_end)
   {
      i++;
      time.Get_SystemTime();
      if(time >= startTime) flag_end = true;
   }
   (*tmp_trace) << trax.GetStream();
}

int main() {

   ModeManager manager;
   TraceString trace;
   int periodo = 100000;
   int vcet = 10000;

   // Creazione dei Thread
   ModeThread th1(boost::bind(function, &trace, "A" ,vcet), periodo);
   ModeThread th2(boost::bind(function, &trace, "B" ,vcet*2), periodo*2);
   ModeThread th3(boost::bind(function, &trace, "C" ,vcet*4), periodo*4);
   ModeThread th4(boost::bind(function, &trace, "D" ,vcet*8), periodo*8);
   ModeThread th5(boost::bind(function, &trace, "E" ,vcet*16), periodo*16);
   ModeThread th6(boost::bind(function, &trace, "F" ,vcet*32), periodo*32);

   // Imposto la Mappa dei Modi
   manager.Recorder(&th1, UNO);
   manager.Recorder(&th4, UNO);
```

```
    manager.Recorder(&th2, DUE);
    manager.Recorder(&th3, DUE);
    manager.Recorder(&th5, DUE);
    manager.Recorder(&th6, DUE);

    manager.Recorder(&th1, TRE);
    manager.Recorder(&th2, TRE);
    manager.Recorder(&th3, TRE);
    manager.Recorder(&th4, TRE);
    manager.Recorder(&th5, TRE);
    manager.Recorder(&th6, TRE);

    // Start dei Mode Thread
    cout << "Start" << endl;
    manager.Start();

    cout << "Thread in Esecuzione: "
         << ModesSingleton::Instance().Get_NumberExecutionThread() << endl;

  SLEEP(1);
  cout << "Set to Mode 1" << endl;
  ModesSingleton::Instance().Set_ActualMode(UNO);

  SLEEP(4);
  cout << "Set to Mode 2" << endl;
  ModesSingleton::Instance().Set_ActualMode(DUE);

  SLEEP(3);
  cout << "Set to Mode 3" << endl;
  ModesSingleton::Instance().Set_ActualMode(TRE);

  SLEEP(1);
  cout << "Set to Mode 4" << endl;
  ModesSingleton::Instance().Set_ActualMode(QUATTRO);

  SLEEP(1);
  cout << "Activate all Thread" << endl;
  ModesSingleton::Instance().Set_ActualMode(ON_ALL);

  SLEEP(1);

    cout << "Exit" << endl;
    manager.Exit();
    return 0;
}
```

L'applicazione KernelShark restituisce il seguente grafico:

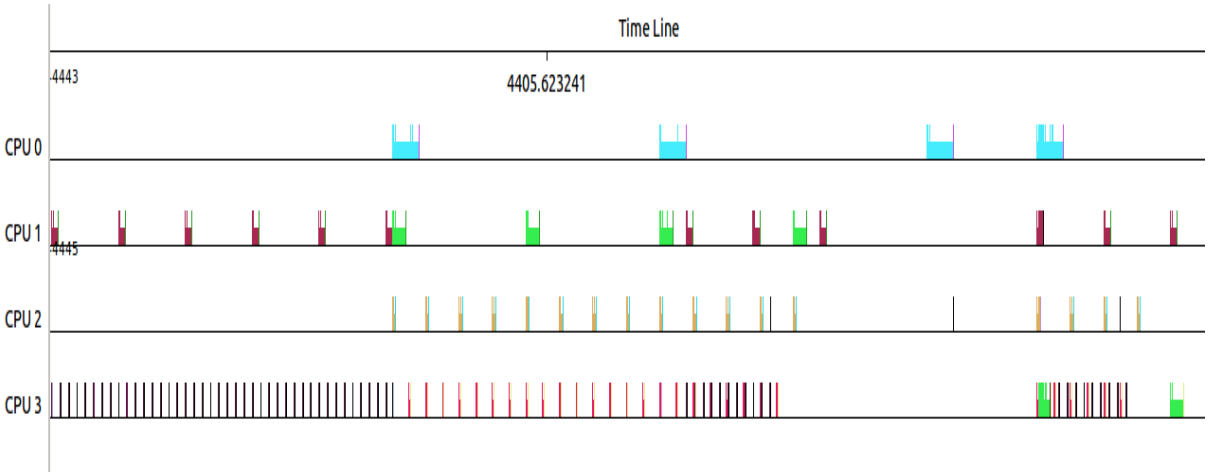


Figura 33. Traccia del Kernel in relazione all'esecuzione dei cambi di modo di lavoro dei task

dove riusciamo a distinguere in modo molto semplice sia i cambi di modo che le fasi transitorie come abbiamo evidenziato di seguito:

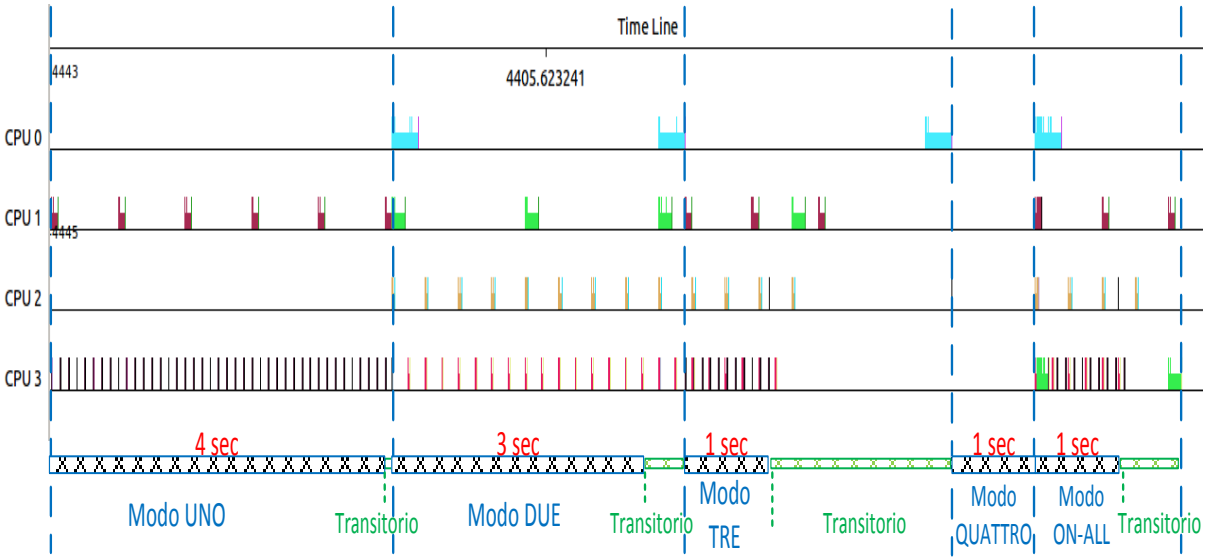


Figura 34. Individuazione nella traccia del Kernel dei Modi e dei transitori durante l'esecuzione dei cambi di modo

In questo secondo esperimento abbiamo potuto verificare la correttezza del funzionamento dei cambi di modo, anche in questo caso possiamo notare la semplicità con cui abbiamo potuto realizzare e controllare questo complicato meccanismo. Come possiamo notare e come ci aspettavamo, la nostra realizzazione pecca nella fase transitoria del cambiamento di modo, risultando carente nella prontezza di reazione del sistema: nel caso in cui il periodo di esecuzione di uno dei task del vecchio modo sia molto più grande rispetto agli altri, prima di poter mandare in esecuzione i task del nuovo modo, dobbiamo attendere che questo venga eseguito sia nel caso in cui continui la propria esecuzione, sia nel caso in cui questo debba sospendersi.

Cap. 7 - Conclusioni

Per dare un giudizio del lavoro svolto, baseremo la nostra analisi sui risultati ottenuti nella fase sperimentale. Dal primo esperimento, in cui eseguiamo concorrentemente 5 thread, possiamo notare che i tempi di setup delle due implementazioni, una con la libreria sviluppata e una senza, sono circa equivalenti. Questo era uno degli obiettivi che ci eravamo prefissati, non volevamo sviluppare un metodo più veloce di quelli esistenti, ma un metodo che ci permettesse di realizzare e gestire i thread periodici in modo agevole e sbrigativo: questo obiettivo è stato evidentemente realizzato. E' stato semplicissimo anche rivisitare la specializzazione della PeriodicRTThread facilitandone l'utilizzo: abbiamo quindi realizzato anche l'aspettativa della riusabilità e dello sviluppo in senso Object Oriented.

Anche il meccanismo dei cambi di modo utilizzato nel secondo esperimento denota la semplicità con cui è stato implementato e controllato questo complicato meccanismo. Questa nostra implementazione, come ci aspettavamo, pecca nella fase transitoria del cambiamento di modo, risultando carente nella prontezza di reazione del sistema.

Aggiungiamo che gli esperimenti svolti sono stati eseguiti sia su macchina Linux che su macchina Windows, abbiamo quindi rispettato anche l'aspettativa della trasportabilità. A fine progetto, abbiamo aggiunto la possibilità di schedulare con politica EDF su sistema Linux con Kernel Modificato. Anche questa successiva modifica è risultata molto semplice, denotando anche in questa occasione la bontà della progettazione Object Oriented della Libreria.

Parallelamente alla fase di sviluppo, oltre agli esperimenti, abbiamo realizzato minuziosamente delle fasi di test su tutte le funzionalità della libreria, ciò ha permesso e permette, nel caso di successive modifiche, di verificare la coerenza e la compatibilità rispetto a quanto precedentemente realizzato: sono stati implementati 31 test.

Possiamo quindi affermare che gli obiettivi che ci eravamo prefissati sono stati tutti rispettati, e che la libreria realizzata può essere certamente considerata un valido aiuto nella realizzazione di applicazioni basate su sistemi real-time; la libreria ThreadUtility è quindi uno strumento all'avanguardia con gli attuali standard di mercato e speriamo susciterà interesse da parte degli sviluppatori.

Appendice A - Tipologia Classi dei Caratteri nelle Regolar Expression

Di seguito sono riportati i nomi delle classi di caratteri supportati dalla Boost::Regex

Nome	NOME STANDARD POSIX	Descrizione
alnum	SI	Qualsiasi carattere alfanumerico.
alpha	SI	Qualsiasi carattere alfabetico.
blank	SI	Qualsiasi carattere di spaziatura a patto che non sia un separatore di riga.
cntrl	SI	Qualsiasi carattere di controllo.
d	NO	Qualsiasi cifra decimale
digit	SI	Qualsiasi cifra decimale.
graph	SI	Qualsiasi carattere grafico.
l	NO	Qualsiasi carattere minuscolo.
lower	SI	Qualsiasi carattere minuscolo.
print	SI	Qualsiasi carattere stampabile.
punct	SI	Qualsiasi carattere di punteggiatura.
s	NO	Ogni carattere di spaziatura.
space	SI	Ogni carattere di spaziatura.
unicode	NO	Qualsiasi carattere il cui codice esteso sia superiore al valore 255.
u	NO	Qualsiasi carattere maiuscolo.
upper	SI	Qualsiasi carattere maiuscolo.
w	NO	Qualsiasi parola di caratteri (caratteri alfanumerici più il carattere di sottolineatura).
word	NO	Qualsiasi parola di caratteri (caratteri alfanumerici più il carattere di sottolineatura).
xdigit	SI	Qualsiasi carattere che sia cifra esadecimale.

Appendice B - KernelShark

Il tool viene utilizzato per rendere più leggibile il buffer generato dalla Ftrace's[9] (12). La traccia del kernel dovrà essere registrare nel file trace.dat e successivamente KernelShark riuscirà ad interpretarla in modo grafico.

Ogni trama di linea raffigurata rappresenta una CPU on-line. Ogni attività sarà rappresentata con un colore diverso. Le linee che fuoriescono dalla parte superiore delle barre rappresentano gli eventi che appaiono nella lista sottostante.

```
# trace-cmd record -e 'sched_wakeup*' -e sched_switch -e 'sched_migrate*' migrate
# kernelshark
```

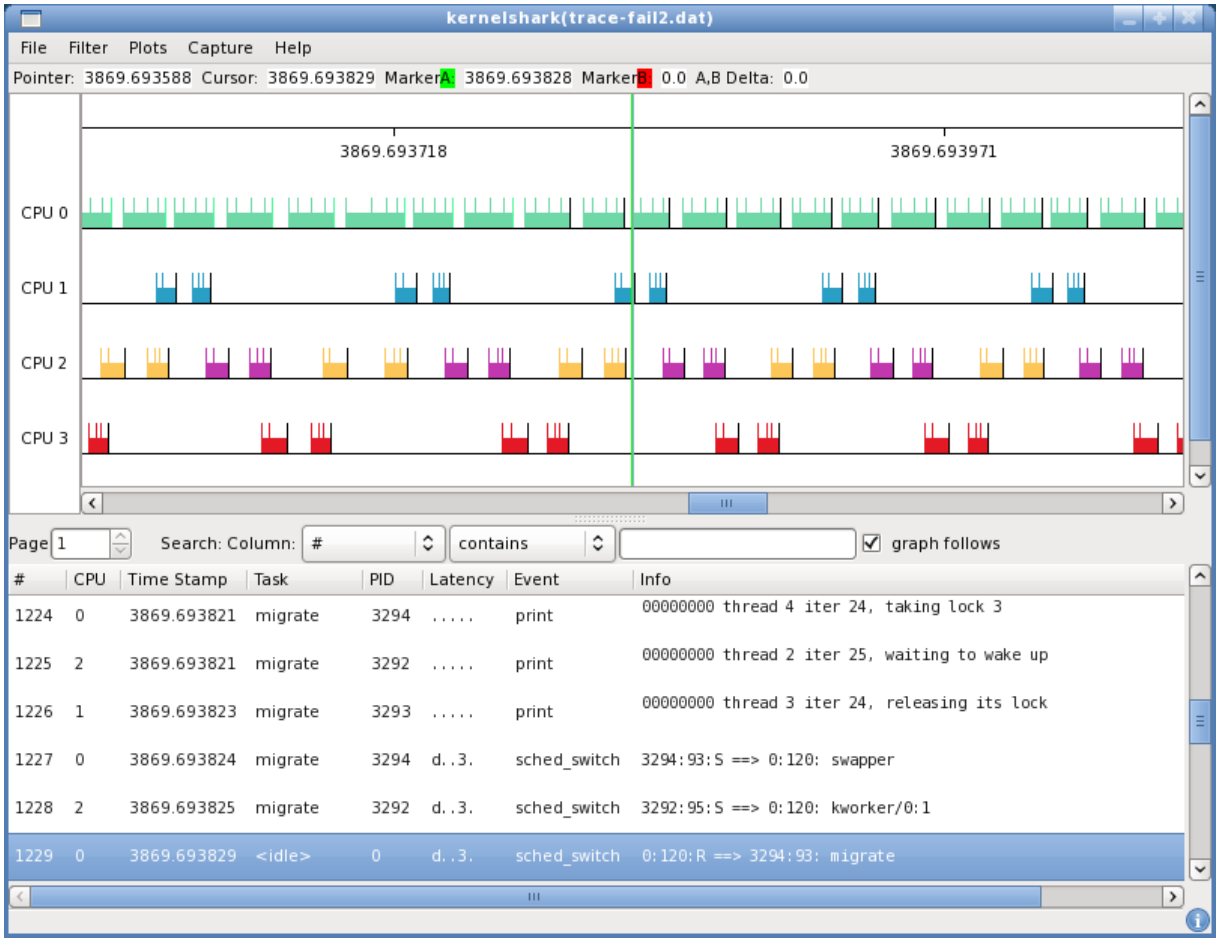


Figura 35. KernelShark

Per approfondimenti su internet è possibile trovare dei casi di esempio (10) (11).

[9] Ftrace è un tracer designato per aiutare gli sviluppatori e i progettisti di sistemi per capire che cosa sta accadendo all'interno del kernel. Esso può essere utilizzato per debuggare o analizzare le latenze e le prestazioni che avvengono al di fuori dello spazio utente.
Anche se ftrace è una funzione di tracer, essa include una infrastruttura di memorizzazione che permette altri tipi di tracciamento. Alcuni degli elementi traccianti che sono attualmente nel ftrace includono un elemento tracciante per seguire le interruzioni di contesto, durante il tempo che viene richiesto affinché un'operazione prioritaria funzioni dopo che è stato svegliato, le interruzioni sono disabilitate, e più.

Indice delle Figure

Bibliografia

1. **Pedro, P.** *Schedulabilità of mode changes in flexible real-time distributed systems.* 1999.

2. Posix Thread Programming. [Online] https://computing.llnl.gov/tutorials/pthreads/.

3. Boost C++ Libraries. [Online] http://www.boost.org/.

4. Google Test. [Online] http://code.google.com/p/googletest/.

5. **Gamma, Erich, et al., et al.** *Design Patterns.* 1995.

6. SCHED_DEADLINE. *Evidence.* [Online] http://www.evidence.eu.com/content/view/313/390/.

7. SCHED_DEADLINE. *Gitorious.* [Online] http://gitorious.org/sched_deadline.

8. **Lelli, Juri.** *Design and development of deadine based scheduling mechanisms for multiprocessor systems.* 2010.

9. **Real, Jorge and Crespo, Alfons.** *Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal.* 2004.

10. KernelShark. [Online] http://rostedt.homelinux.com/kernelshark/.

11. KernelShark. [Online] http://lwn.net/Articles/425583/.

12. FTrace. [Online] http://lwn.net/Articles/322666/.

13. **Buttazzo, Giorgio.** *Hard Real-Time Computing Systems.* 2011.

14. MultiThreaded Programming (POSIX pthreads Tutorial). [Online] http://randu.org/tutorials/threads/.